OpenFF Toolkit Documentation

Release 0.16.0+3.g53db31f.dirty

Open Force Field Consortium

Apr 29, 2024

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 mamba

The simplest way to install the Open Force Field Toolkit is via Mamba, a drop-in replacement for the Conda package manager. We publish packages via `conda-forge`. With Mamba installed, use it to install the OpenFF Toolkit into a new environment:

```
$ mamba create -n openff-toolkit -c conda-forge openff-toolkit
```

To use the new environment in a shell session, you must first activate it:

```
$ mamba activate openff-toolkit
```

If you do not have Mamba or Conda installed, see the ecosystem installation documentation.

Note: Installation via the Mamba package manager is the recommended 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 (x86) MacOS and Ubuntu Linux.

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 `pip`:

```
$ cd openff-toolkit
$ python -m pip install .
```

1.3 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 "built-in" 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.

1.3.1 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.

1.3.2 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.

1.3.3 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:

```
$ mamba install -c openeye 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.

1.3.4 Check installed toolkits

All available toolkits are automatically registered in the `GLOBAL_TOOLKIT_REGISTRY`. The available toolkits and their versions can be inspected through the `registered_toolkit_versions` dictionary attribute:

```
from openff.toolkit import GLOBAL_TOOLKIT_REGISTRY
print(GLOBAL_TOOLKIT_REGISTRY.registered_toolkit_versions)
# {'The RDKit': '2022.03.5', 'AmberTools': '22.0', 'Built-in Toolkit': None}
```

CHAPTER

TWO

EXAMPLES USING SMIRNOFF WITH THE TOOLKIT

For users: These examples are best viewed at our centralized documentation page, where you will find a variety of ways to view and run the notebooks.

For developers: The following examples live in the OpenFF toolkit repository.

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
- vsite_showcase - two examples of using SMIRNOFF force fields virtual sites - a ligand with virtual sites on sulfure and chlorine and a comparison between the SMIRNOFF and OpenMM implementations of TIP5P
- QCArchive_interface - retrieving data from the QCArchive into `Molecule` objects
- 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)
- 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 Current development

3.1.1 API-breaking changes

3.1.2 Behavior changes

3.1.3 Bugfixes

3.1.4 New features

3.1.5 Improved documentation and warnings

3.2 0.16.0

3.2.1 Behavior changes

- PR #1852: Fixes issue #1363: If multiple parameters have identical SMIRKS, calls to `ParameterHandler[smirks]` and `ParameterList[smirks]` will now return the LAST parameter with a matching SMIRKS. Previously this would return the FIRST parameter with a matching SMIRKS, which was a confusing behavior given SMIRNOFF hierarchy rules.

3.2.2 Bugfixes

- PR #1846: Band-aids issue #1842, where the Toolkit Showcase example would sporadically fail when run with one core.

3.2.3 New features

- PR #1827: Adds `Topology.clear_positions`
- PR #1852: Adds the `allow_duplicate_smirks` named argument to `ParameterHandler.add_parameter`. Previously it was possible to make a force field with duplicate SMIRKS by loading it from file or combining multiple FFs, so this also lets you do it using the API.
- PR #1826: Allow writing molecules to `Paths`
- PR #1797: `Topology.from_pdb` can now load more file-like objects, including `io.StringIO`.
- PR #1808: Improves default representation of ValenceDict.
- PR #1834: Adds `Molecule.get_available_charge_methods` and `BaseWrapper.supported_charge_methods`.
- PR #1837: Decouples vdW and virtual site parameters.

3.2.4 Improved documentation and warnings

- PR #1795: Add `NAGLToolkitWrapper` to API reference
- PR #1796: Update docs tooling and fix warnings
- PR #1786: Describe contributing to documentation notebooks in developers guide.
- PR #1845: Update convert_all_strings_to_quantity docstring.
- PR #1849: Add "how are charges assigned?" section to FAQ.
- PR #1798: Adds type annotations to most of the codebase.

3.2.5 Tests updated

- PR #1836: Update tests to not directly call `numpy.random.random` any more (per NEP 19)

3.3 0.15.2

3.3.1 New features

- PR #1818: Several improvements to enable the practical use of `Interchange.from_openmm`.

3.4 0.15.1

3.4.1 Tests updated

- PR #1814: Fixes a test to be compatible with both pydantic 1 and 2.

3.5 0.15.0

This release adds compatibility with QCFractal >=0.50.0, but removes compatibiltiy with QCFractal <0.50.0.

3.5.1 API-breaking changes

- PR #1760: Removes the private, unused `ParameterHandler._OPENMMTYPE` attribute.
- PR #1763: Updates the OpenFF Toolkit to be compatible with QCFractal >= 0.50. Removes the `client` named argument from `Molecule.from_qcschema`.

3.5.2 Behavior changes

- PR #1779: Overhauls `Molecule.enumerate_protomers`:
  - By default, `max_states=0` and there is no limit to the number of returned protomers
  - The input molecule is returned if it is enumerated with default Quacpac formal charge enumeration settings

3.5.3 Bugfixes

- PR #1778: Ensures SD data tags are preserved in `Molecule.from_openeye` if the input is of type `oechem.OEGraphMol`.
- PR #1811: Preserves hierarchy data in `_SimpleMolecule.from_molecule`

3.5.4 New features

- PR #1775: Re-exports `openff.units.unit` and `Quantity` at `openff.toolkit.unit` and `Quantity`.
- PR #1805: Adds `_SimpleMolecule.__deepcopy__` and `_SimpleMolecule.to_topology`.

3.5.5 Improved documentation and warnings

- PR #1732: Add documentation describing the use of PDB files with the toolkit.
- PR #1804: Makes `ForceField.parse_sources` docstring consistent with implementation.

3.6 0.14.5

3.6.1 Bugfixes

- PR #1740: Updates for Mypy 1.6.
- PR #1749: Updates versioneer for Python 3.12 compatibility.
- PR #1756: Fixes issue #1739, where virtual sites would be double-created under some circumstances.

3.6.2 New features

- PR #1731: Support SMIRNOFF vdW version 0.5.

3.6.3 Improved documentation and warnings

- PR #1747: Warns if a SMILES with full atom mappings is passed to `Molecule.from_smiles`, which does not use the atom map for atom ordering (`Molecule.from_mapped_smiles` does).
- PR #1743: Uses a longer stride in OpenMM DCD reporter in the toolkit showcase and should better utilize GPU resources, if available.
- PR #1744: Updates the virtual site notebook to use new Interchange behavior.

3.7 0.14.4

3.7.1 Behavior changes

- PR #1705: Do not raise warning when `allow_undefined_stereo=True`.
- PR #1695: `ChemicalEnvironmentParsingError` is now raised when an underlying toolkit fails to parse a SMARTS/SMIRKS pattern it is given during substructure matching.
- PR #1716: Adds deprecation warnings to `Molecule.from_polymer_pdb`, `Molecule.from_pdb_and_smiles`, and `RDKitToolkitWrapper.from_pdb_and_smiles` instead pointing users toward `Topology.from_pdb`.

3.7.2 Bugfixes

- PR #1726: Fix error message in setting `scale12`, `scale13`, and `scale15` attributes.
- PR #1728: Adds support for loading deprotonated cysteine (sometimes labeled as "CYM") residues in `Topology.from_pdb`

3.7.3 New features

- PR #1733: Makes `NAGLToolkitWrapper` and its associated module public.
- PR #1698: Makes `openff.toolkit.utils.toolkit_registry.toolkit_registry_manager` public.
- PR #1662: Adds hierarchy scheme iterators to `Topology`, i.e. `Topology.residues`, when schemes of the same iterator name are defines on all constituent `Molecules`.
- PR #1700: Use `openff-nagl` v0.3.0.
- PR #1623: Adds `Topology.visualize`.

3.7.4 Improved documentation and warnings

- PR #1709: Update molecule cookbook to use the maximally capable `Topology.from_pdb` in lieu of the more limited `Molecule.from_pdb_and_smiles` and `Molecule.from_polymer_pdb`.
- PR #1719: Remove out-of-date and unused `examples/environment.yaml` and various examples updates.

3.8 0.14.3

3.8.1 Bugfixes

- PR #1689: Fixes #1688 in which automatic up-conversion of version 0.3 of `vdWHandler` created via the Python API errored out if `method` was not specified.
- PR #1690: Fixes a circular-import bug that occurs when attempting to print a "no cheminformatics toolkits available" warning.

3.9 0.14.2

3.9.1 Behavior changes

- PR #1679: Version 0.3 `<vdW>` sections of OFFXML files will automatically be up-converted (in memory) to version 0.4 according to the recomendations provided in OFF-EP 0008.

3.9.2 New features

- PR #1631: Adds support for Python 3.11.

3.10 0.14.1

3.10.1 API-breaking changes

- PR #1664: Removes `ChemicalEnvironment` and the entire `openff.toolkit.typing.chemistry` submodule, which was deprecated in 0.12.0.

3.10.2 Behavior changes

- PR #1675: Makes InChI parsing failures more informative and gives them their own exception type, `InChIParseError`.

3.10.3 New features

- PR #1627: (beta release of major new feature by @connordavel) Adds experimental support for custom substructure loading in `Topology.from_pdb`, via the `_custom_substructures` keyword argument. This will be added to the public API (by removing the leading underscore) in a future feature release, but is available for testing now. This feature should allow for easier loading of modified amino acids, nucleic acids, and other polymers.

3.10.4 Bugfixes

- PR #1662: Fixes issue #1660 by forbidding the registration of `HierarchySchemes` with iterator names that conflict with existing `Molecule` attributes.
- PR #1667: A more helpful exception is now raised when `Topology.from_openmm` is given an OpenMM Topology with virtual sites.

3.10.5 Examples updated

- PR #1671: Re-rendered all examples using RDKit+AmberTools backend, and using most recent version of OFF Toolkit.

3.10.6 Improved documentation and warnings

3.11 0.14.0

3.11.1 API-breaking changes

- PR #1649: Removes tests and associated modules from the public API.
- PR #1508: Removes the `return_topology` kwawrg from `ForceField.create_openmm_system` which was deprecated in version 0.11.0. To access the `Topology` that results from parametrization, call `ForceField.create_interchange` and access the `.topology` attribute of the returned `Interchange` object.
- PR #1506: Removes several classes and properties in the `topology` submodule that were deprecated in version 0.11.0.

3.11.2 Behavior changes

- PR #1652: Fixes issue #1642 by making AmberToolsToolkitWrapper thread-safe (previously `AmberToolsToolkitWrapper.assign_partial_charges` and `assign_fractional_bond_orders` were not)

3.11.3 Bugfixes

- PR #1654: Fixes issue #1653, where a test that expected RDKit to fail began returning an error when RDKit became able to generate conformers for octahedral molecules.

3.11.4 Improved documentation and warnings

- PR #1572: Improved installation guide in line with ecosystem docs.

3.11.5 Examples updated

- PR #1644: Streamlines several examples by using `Interchange.to_openmm_simulation`.
- PR #1651: Fix broken links in several examples.
- PR #1648: Update examples to use `Topology.from_pdb` and other small cleanups.

3.12 0.13.2

3.12.1 Bugfixes

- PR #1640: Fixes issue #1633 in which some force field attributes were erroneously parsed as `Quantity` objects and issue #1635 in which OpenFF 2.1.0 ("Sage") could not be loaded with Pint 0.22.

3.12.2 Improved documentation and warnings

- PR #1636 and PR #1643: Make the Molecule Cookbook and `Molecule.from_qcschema` docstring only pull down QCF records with fully defined stereo.

3.13 0.13.1

3.13.1 Behavior changes

- PR #1619: Fixes **silent error** #1618, by making partial_charges.setter more comprehensive in type and shape checking.

3.13.2 Bugfixes

- PR #1617: Fixes #1616, by converting NAGL charges to float and converting partial_charges to float before converting to_openeye()
- PR #1622: Fixes #1621 and #1346 in which some file-loading methods would fail on `pathlib.Path`.

3.14 0.13.0

3.14.1 New features

- PR #1567: Allows setting `Molecule.name` in `Molecule.from_smiles`, `from_inchi`, `from_polymer_pdb`, and `from_pdb_and_smiles`.
- PR #1565: Adds `Topology.from_pdb`.

3.14.2 Behavior changes

- PR #1569: Several instances of `Exception` being raised are now replaced with other exceptions being raised.
- PR #1577: Drops support for Python 3.8, following NEP-29.

3.14.3 Bugfixes

- PR #1589: Fixes Issue #1579, where Molecule.from_polymer_pdb could not handle NH2 caps at C termini.
- PR #1591: Fixes #1563, where `from_rdkit` would sometimes raise an error about radicals if a molecule using a non-MDL aromaticity model was provided.

3.14.4 Improved documentation and warnings

- PR #1564: Improve documentation of conformer selection in `Molecule.assign_partial_charges()`
- PR #1574: Fix class method signature rendering throughout API docs
- PR #1584: Update some outdated docstrings and add some annotations.

3.14.5 Examples updates

- PR #1575: The Toolkit Showcase example has been simplified via use of `Topology.from_pdb`

3.15 0.12.1

3.15.1 New features

- PR #1502: Adds Gasteiger charge computation using the RDKit backend.
- PR #1498: `Molecule.remap()` now supports partial mappings with the `partial` argument.
- PR #1528: `Topology.box_vectors` are can now be set with `openmm.unit.Quantitys`, which are internally converted.

3.15.2 Behavior changes

- PR #1498: New, more complete, and more descriptive errors for `Molecule.remap()`.
- PR #1525: Some unreleased force fields previously accessible from `"openff/toolkit/data/test_forcefields/"` are no longer implicitly available to the `ForceField` constructor.
- PR #1545: Replaced the logic that sorts `HierarchyElements` with dedicated code in the OpenFF Toolkit instead of relying on deprecated features in the `packaging` module.

3.15.3 Bugfixes

- PR #1543: Fixes a bug in which plugins are not loaded if a `ForceField` is constructed prior without plugins.

3.15.4 Improved documentation and warnings

- PR #1498: Improved documentation for `Molecule.remap()`, `Molecule.from_smiles()`, and `Molecule.from_mapped_smiles()`, emphasizing the relationships between these methods. In particular, the documentation now clearly states that `from_smiles()` will not reorder atoms based on SMILES atom mapping.
- PR #1525: Improves reporting failures when loading force fields.
- PR #1513: Improves error messages and documentation around supported aromaticity models (currently only "OEAroModel_MDL").

3.16 0.12.0

3.16.1 New features

- PR #1484: A `positions` argument has been added to `Topology.from_openmm()` and `Topology.from_mdtraj()`, which allows the topology's positions to be set more conveniently.
- PR #1468: Track which `ParameterHandlers` are loaded as plugins.

3.16.2 Behavior changes

- PR #1481: Removes `compute_partial_charges_am1bcc`, which was deprecated in 0.11.0.
- PR #1466: Replaces the use of `collections.OrderedDict` throughout the toolkit with the built-in `dict`. `attach_units`, `detach_units`, and `extract_serialized_units_from_dict` have been removed from `openff.toolkit.utils.utils`.
- PR #1472: Removes `ParameterHandler._VALENCE_TYPE` and the same attribute of its subclasses, which were previously not used. Also deprecates `ChemicalEnvironment` and, by extension, the `openff.toolkit.typing.chemistry` submodule.

3.16.3 Bugfixes

- PR #1476: Fixes #1475 by also registering a `ParameterHandler`'s class when calling `ForceField.register_parameter_handler`.
- PR #1480: Fixes #1479 by requiring that `Atom.atomic_number` is a positive integer.
- PR #1494: Fixes #1493 in which some OFFXML file contents were parsed to `unit.Quantity` objects despite not representing physical quantities.

3.16.4 Improved documentation and warnings

- PR #1484: The docstrings for `Topology.from_openmm()` and `Topology.from_mdtraj()` have been improved.
- PR #1483: Simplified and clarified errors and warnings related to undefined stereochemistry with RDKit.

3.17 0.11.4 Bugfix release

3.17.1 Behavior changes

- PR #1462: Makes residue numbers added by `Molecule.perceive_residues` strings (previously they were ints), to match the behavior of `Topology.from_openmm` and other hierarchy info-setting methods.

3.17.2 Bugfixes

- PR #1459: Fixes #1430, where `Topology.from_openmm` would mis-assign atom names (and probably also hierarchy metadata as well).
- PR #1462: Fixes #1461, where the default `Molecule.residues` iterator wouldn't sort by residue number correctly when residue information was added by `Molecule.perceive_residues`.

3.18 0.11.3 Bugfix release

- PR #1460: Disables error causing Issue #1432, where `Molecule.from_polymer_pdb` would sometimes issue stereochemistry errors when reading valid PDBs using the RDKit backend.

3.18.1 Bugfixes

- PR #1436: Fix a small bug introduced in 0.11.2, where running with OpenEye installed but not licensed could lead to a crash.
- PR #1444: Update for pint 0.20.

3.18.2 Examples updates

- PR #1447: Fixed units of tolerance used in OpenMM minimization in Toolkit Showcase example notebook (from @ziyuanzhao2000)

3.18.3 Improved documentation and warnings

- PR #1442: Doctests added to CI, leading to numerous fixed docstrings and examples therein.

3.18.4 Miscellaneous

- PR #1413: Remove some large and unused data files from the test suite.
- PR #1434: Remove dependency on `typing_extensions`.

3.19 0.11.2 Bugfix release

3.19.1 Behavior changes

- PR #1421: Allow `Molecule.from_rdkit()` to load D- and F- block radicals, which cannot have implicit hydrogens.

3.19.2 Bug fixes

- PR #1417: Ensure the properties dict is copied when a `Molecule` is.

3.19.3 Improved documentation and warnings

- PR #1426: A warning about OpenEye Toolkits being unavailable is only emitted when they are installed but the license file is not found.

3.20 0.11.1 Minor release forbidding loading radicals

3.20.1 Behavior changes

- PR #1398: Updates the `Bond.bond_order` setter to only accept int values.
- PR #1236: `from_rdkit` and `from_openeye` now raise an `RadicalsNotSupportedError` when loading radicals. It's not clear that the OpenFF Toolkit was ever safely handling radicals - they appear to be the root cause of many instances of unintended hydrogen addition and other connection table changes. If this change affects a workflow that was previously working correctly, please let us know on this issue so we can refine this behavior.

3.20.2 Examples changed

- PR #1236: `examples/check_dataset_parameter_coverage` has been deprecated.

3.20.3 Bug fixes

- PR #1400: Fixes a bug where `Molecule.from_pdb_and_smiles` could incorrectly order coordinates.
- PR #1404: Support default hierarchy schemes in outputs of `Molecule.from_pdb_and_smiles()` and `Topology.from_openmm()`

3.21 0.11.0 Major release adding support for proteins and refactoring the Topology class.

3.22 Migration guide

3.22.1 New `Molecule.from_polymer_pdb()` method for loading proteins from PDB files

The Toolkit now supports loading protein PDB files through the `Molecule.from_polymer_pdb()` class method. For now, PDB files must consist of only a single protein molecule composed only of the 20 standard amino acids, their common protonated and deprotonated conjugates, and the N-methyl and acetyl caps.

3.22.2 Important API points re-exported from `openff.toolkit`

A number of commonly used API points have been re-exported from the package root. This should make using the Toolkit simpler for most people. The previous API points remain available. These API points are lazy-loaded so that parts of the toolkit can still be loaded without loading the entire code-base.

The most important of these are the `ForceField`, `Molecule`, and `Topology` classes:

```
- from openff.toolkit.typing.engines.smirnoff import ForceField
- from openff.toolkit.topology import Molecule, Topology
+ from openff.toolkit import ForceField, Molecule, Topology
```

A number of other useful API points are also available through this mechanism:

```
- from openff.toolkit.typing.engines.smirnoff import get_available_force_fields
- from openff.toolkit.utils.toolkits import (
-     GLOBAL_TOOLKIT_REGISTRY,
-     AmberToolsToolkitWrapper,
-     BuiltInToolkitWrapper,
-     OpenEyeToolkitWrapper,
-     RDKitToolkitWrapper,
-     ToolkitRegistry,
- )
+ from openff.toolkit import (
+     get_available_force_fields,
+     GLOBAL_TOOLKIT_REGISTRY,
+     AmberToolsToolkitWrapper,
+     BuiltInToolkitWrapper,
+     OpenEyeToolkitWrapper,
+     RDKitToolkitWrapper,
+     ToolkitRegistry,
+ )
```

The `topology`, `typing`, and `utils` modules can also be lazy loaded after importing only the top-level module:

```
- import openff.toolkit.topology
+ import openff.toolkit
  atom = openff.toolkit.topology.Atom()
```

3.22.3 Units

The use of OpenMM units has been replaced by the new OpenFF Units package, based on Pint.

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.m_as(unit.angstrom))  # Note that value.magnitude_as() does not exist
# 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')>
```

Breaking change: Removal of `openff.toolkit.utils.check_units_are_compatible()`

The `openff.toolkit.utils.check_units_are_compatible()` function has been removed. Use `openff.units.Quantity.is_compatible_with()` and `openff.units.openmm.from_openmm()` instead:

```
- check_units_are_compatible("length", length, openmm.unit.angstrom)
+ from_openmm(length).is_compatible_with(openff.units.unit.angstrom)
```

3.22.4 Breaking change: Interchange now responsible for system parametrization

Code for applying parameters to topologies has been removed from the Toolkit. This is now the responsibility of OpenFF Interchange. This change improves support for working with parametrized systems (through the `Interchange` class), and adds support for working with simulation engines other than OpenMM.

The `ForceField.create_interchange()` method has been added, and the `ForceField.create_openmm_system()` method now uses Interchange under the hood.

As part of this change, the `UnsupportedKeywordArgumentsError` has been removed; passing unknown arguments to `create_openmm_system` now raises a `TypeError`, as is normal in Python.

The following classes and methods have been **removed** from `openff.toolkit.typing.engines.smirnoff.parameters`:

- `NonintegralMoleculeChargeException`
- `NonbondedMethod`

- `ParameterHandler.assign_parameters()`
- `ParameterHandler.postprocess_system()`
- `ParameterHandler.check_partial_bond_orders_from_molecules_duplicates()`
- `ParameterHandler.assign_partial_bond_orders_from_molecules()`

In addition, the `ParameterHandler.create_force()` method has been deprecated and its functionality has been removed. It will be removed in a future release.

The `return_topology` argument of `create_openmm_system` has also been deprecated, and will be removed in 0.12.0. To create an OpenMM topology, use `Interchange`:

```
- omm_sys, off_top = force_field.create_openmm_system(
-     topology,
-     return_topology=True,
- )
- omm_top = off_top.to_openmm()
+ interchange = force_field.create_interchange(topology)
+ omm_sys = interchange.to_openmm(combine_nonbonded_forces=True)
+ omm_top = interchange.to_openmm_topology()
```

If you need access to the modified OpenFF topology for some other reason, create an `Interchange` and retrieve it there:

```
- omm_sys, off_top = force_field.create_openmm_system(
-     topology,
-     return_topology=True,
- )
+ interchange = force_field.create_interchange(topology)
+ off_top = interchange.topology
+ omm_sys = interchange.to_openmm(combine_nonbonded_forces=True)
```

3.22.5 Breaking change: `Topology` molecule representation

`Topology` objects now store complete copies of their constituent `Molecule` objects, rather than using simplified classes specific to `Topology`. This dramatically simplifies the code base and allows the use of the full `Molecule` API on molecules inside topologies.

The following classes have been **removed**:

- `TopologyAtom` (use `Atom` instead)
- `TopologyBond` (use `Bond` instead)
- `TopologyMolecule` (use `Molecule` instead)

The following properties have been **deprecated** and will be removed in a future release:

- `Topology.n_topology_atoms` (use `Topology.n_atoms` instead)
- `Topology.topology_atoms` (use `Topology.atoms` instead)
- `Topology.n_topology_bonds` (use `Topology.n_bonds` instead)
- `Topology.topology_bonds` (use `Topology.bonds` instead)
- `Topology.n_topology_particles` (use `Topology.n_particles` instead)
- `Topology.topology_particles` (use `Topology.particles` instead)

- `Topology.reference_molecules` (use `Topology.unique_molecules` instead)
- `Topology.n_reference_molecules` (use `Topology.n_unique_molecules` instead)
- `Topology.n_topology_molecules` (use `Topology.n_molecules` instead)
- `Topology.topology_molecules` (use `Topology.molecules` instead)
- `Topology.n_particles` (use `Topology.n_atoms` instead)
- `Topology.particles` (use `Topology.atoms` instead)
- `Topology.particle_index` (use `Topology.atom_index` instead)

In addition, the `Topology.identical_molecule_groups` property has been added, to facilitate iterating over copies of isomorphic molecules in a `Topology`.

3.22.6 Breaking change: Removed virtual site handling from topologies

To maintain a clear distinction between a model and the chemistry it represents, virtual site handling has been removed from the Toolkit's `Topology` and `Molecule` classes. Virtual site support remains in the force field side of the toolkit, but creating virtual sites for particular molecules is now the responsibility of OpenFF Interchange. This allows the same `Topology` to be used for force fields that use different virtual sites; for example, a topology representing a solvated protein might be parametrized separately with a 3-point and 4-point water model.

As part of this change, the distinction between `Atom` and `Particle` is deprecated. The `Particle` class will be removed in a future release.

The following classes have been **removed**:

- `BondChargeVirtualSite`
- `DivalentLonePairVirtualSite`
- `MonovalentLonePairVirtualSite`
- `TrivalentLonePairVirtualSite`
- `VirtualParticle`
- `VirtualSite`
- `TopologyVirtualParticle`
- `TopologyVirtualSite`

The following methods and properties have been **removed**:

- `Atom.add_virtual_site()`
- `Atom.virtual_sites`
- `FrozenMolecule.compute_virtual_site_positions_from_conformer()`
- `FrozenMolecule.compute_virtual_site_positions_from_atom_positions()`
- `FrozenMolecule.n_virtual_sites`
- `FrozenMolecule.n_virtual_particles`
- `FrozenMolecule.virtual_sites()`
- `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()`
- `Molecule.add_bond_charge_virtual_site()`
- `Topology.n_topology_virtual_sites`
- `Topology.topology_virtual_sites`
- `Topology.virtual_site()`
- `Topology.add_particle()`

The following properties have been **deprecated** and will be removed in a future release:

- `Molecule.n_particles` (use `Molecule.n_atoms` instead)
- `Molecule.particles` (use `Molecule.atoms` instead)
- `Molecule.particle` (use `Molecule.atom` instead)
- `Molecule.particle_index` (use `Topology.n_atoms` instead)

3.22.7 Atom metadata and hierarchy schemes for iterating over residues, chains, etc.

The new `Atom.metadata` attribute is a dictionary that can store arbitrary metadata. Atom metadata commonly includes residue names, residue sequence numbers, chain identifiers, and other metadata that is not essential to the functioning of the Toolkit. Metadata can then be passed on when a `Molecule` is converted to another package; see *Molecule conversion to other packages*.

Metadata can also support iteration through the `HierarchyScheme` class. A hierarchy scheme is defined by some uniqueness criteria. Iterating over the scheme iterates over groups of atoms that have identical metadata values for the defined uniqueness criteria. For more information, see the API docs for `HierarchyScheme` and its related methods.

3.22.8 Breaking change: Removed `Topology.charge_model` and `Topology.fractional_bond_order_model`

Due to flaws in previous versions of the OFF Toolkit, these properties never had an effect on the assigned parameters. To resolve this bug and maintain a clear distinction between a model and the chemistry it represents, the `Topology.charge_model` and `Topology.fractional_bond_order_model` properties have been removed. Charge models and FBOs are now the responsibility of the ForceField.

3.22.9 Breaking change: Removed `Atom.element`

`Atom.element` has been removed to reduce our dependency on OpenMM for core functions:

```
- atomic_number = atom.element.atomic_number
+ atomic_number = atom.atomic_number
- atom_mass = atom.element.mass
+ atom_mass = atom.mass
- atom_elem_symbol = atom.element.symbol
+ atom_elem_symbol = atom.symbol
```

3.22.10 `Topology.to_file()`

The `Topology.to_file()` method has been significantly revised, including three breaking changes.

Breaking change: `filename` argument renamed `file`

The `filename` argument has been renamed `file`, and now supports file-like objects in addition to file names:

```
-  topology.to_file(filename="out.pdb", positions=xyz)
+  topology.to_file(file="out.pdb", positions=xyz)
```

Breaking change: Atom names guaranteed unique per residue by default

The default behavior is now to ensure that atom names are unique within a residue, rather than within a molecule. The `ensure_unique_atom_names` argument has been added to control this behavior. The previous behavior can be achieved by passing `True` to `ensure_unique_atom_names`:

```
- topology.to_file("out.pdb", xyz)
+ topology.to_file("out.pdb", xyz, ensure_unique_atom_names=True)
```

The `ensure_unique_atom_names` argument can also take the name of a `HierarchyScheme`, in which case atom names will be unique within the elements of that scheme (instead of within the atoms of a molecule). If the scheme is missing from a molecule, atom names will be unique within that molecule. The default value of this argument is `"residues"` to preserve atom names from the PDB.

Breaking change: `keepIds` argument renamed `keep_ids`

The `keepIds` argument has been renamed to the more Pythonic `keep_ids`. Its behavior and position in the argument list has not changed.

```
- topology.to_file("out.pdb", xyz, keepIds=True)
+ topology.to_file("out.pdb", xyz, keep_ids=True)
```

Non-breaking changes

In addition to these breaking changes, the `positions` argument is now optional. If it is not provided, positions will be taken from the first conformer of each molecule in the topology. If any molecule has no conformers, an error will be raised.

3.22.11 Positions in topologies

The `Topology.get_positions()` and `Topology.set_positions()` methods have been added to facilitate working with coordinates in topologies. A topology's positions are defined by the zeroth conformer of each molecule. If any molecule lacks conformers, the entire topology has no positions.

3.22.12 Parameter types moved out of handler classes

To facilitate their discovery and documentation, re-exports for the `ParameterType` classes have been added to the `openff.toolkit.typing.engines.smirnoff.parameters` module. Previously, they were accessible only within their associated `ParameterHandler` classes. This is not a breaking change.

```
- from openff.toolkit.typing.engines.smirnoff.parameters import BondHandler
- BondType = BondHandler.BondType
+ from openff.toolkit.typing.engines.smirnoff.parameters import BondType
```

3.22.13 Breaking change: `MissingDependencyError` renamed `MissingPackageError`

The `MissingDependencyError` exception has been renamed `MissingPackageError` to better reflect its purpose.

```
  try:
      ...
- except MissingDependencyError:
+ except MissingPackageError:
      pass
```

3.22.14 `compute_partial_charges_am1bcc()` deprecated

The `compute_partial_charges_am1bcc()` methods of the `Molecule`, `AmberToolsToolkitWrapper` and `OpenEyeToolkitWrapper` classes have been deprecated and will be removed in a future release. Their functionality has been incorporated into `assign_partial_charges()` for more consistency with other charge generation methods:

```
- mol.compute_partial_charges_am1bcc()
+ mol.assign_partial_charges(partial_charge_method='am1bcc')
```

3.22.15 Additional changes and bugfixes

- PR #1105, PR #1195, PR #1301, PR #1331, PR #1322, PR #1372: Add `Molecule.from_polymer_pdb`
- PR #1377: Adds `Topology.unique_molecules`, which largely replaces `Topology.reference_molecules`.
- PR #1313: Fixes Issue #1287, where `OpenEyeToolkitWrapper.assign_partial_charges` didn't request symmetrized charges when the charge model was set to `AM1-Mulliken`.
- PR #1348: Allows `pathlib.Path` objects to be passed to `Molecule.from_file`.
- PR #1276: Removes the `use_interchange` argument to `create_openmm_system`. Deletes the `create_force` and `postprocess_system` methods of `ParameterHandler` 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`.
- PR #1140: Adds the `Topology.identical_molecule_groups` property, which provides a way of grouping the instances of a specific chemical species in the topology.
- 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.
- PR #1346: Conformer generation with RDKit will use `useRandomCoords=True` on a second attempt if the first attempt fails, which sometimes happens with large molecules.
- 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.
- PR #1368: Adds the `Topology.get_positions()` and `Topology.set_positions()` methods for working with topology positions. Positions are represented as the first conformer of each molecule in the topology.
- PR #1368: Allows setting the `ensure_unique_atom_names` argument of `Topology.to_openmm()`to the name of a hierarchy scheme, in which case atom names are guaranteed unique per element of that scheme rather than per molecule. Changes the default value to `"residues"`.
- PR #1368: Adds the `ensure_unique_atom_names` argument to the `Topology.to_file()`, which mimics the same argument in `Topology.to_openmm()`. Renames the `keepIds` argument to `keep_ids`. Renames the `filename` argument to `file` and allows a file-like object to be passed instead of a filename. Makes the `positions` argument optional; if it is not given, positions are take from the first conformer of each molecule in the Topology.
- 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.

3.22.16 Examples added

- PR #1113: Updates the Amber/GROMACS example to use Interchange.
- PR #1368: Updates the Toolkit showcase with the new polymer handling and Interchange support

3.22.17 Tests updated

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

3.23 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.

3.24 0.10.4 Bugfix release

3.24.1 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.

3.25 0.10.3 Bugfix release

3.25.1 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.

3.25.2 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.

3.25.3 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.

3.25.4 Behaviors changed and bugfixes

- PR #1185: Removed length check in ValenceDict and fixed checking the permutations of dihedrals

3.25.5 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.

3.25.6 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.

3.26 0.10.2 Bugfix release

3.26.1 API-breaking changes

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

3.26.2 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.

3.26.3 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.

3.26.4 Examples added

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

3.27 0.10.1 Minor feature and bugfix release

3.27.1 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`.

3.27.2 Tests updated

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

3.27.3 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.

3.28 0.10.0 Improvements for force field fitting

3.28.1 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.

3.28.2 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.

3.28.3 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.

3.28.4 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!

3.28.5 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.

3.29 0.9.2 Minor feature and bugfix release

3.29.1 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`.

3.29.2 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.

3.29.3 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.

3.30 0.9.1 - Minor feature and bugfix release

3.30.1 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.

3.30.2 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.

3.30.3 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.

3.30.4 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 .

3.31 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`.

3.31.1 API-breaking changes

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

3.32 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.

3.33 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.

3.33.1 Bugfixes

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

3.33.2 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.

3.34 0.8.2 - Bugfix release

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

3.34.1 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.

3.35 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.

3.35.1 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.

3.35.2 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.

3.35.3 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.

3.35.4 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)

3.36 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

3.36.1 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`

3.36.2 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` .

3.36.3 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"`.

3.36.4 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.

3.36.5 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`

3.36.6 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

3.37 0.7.2 - Bugfix and minor feature release

3.37.1 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`

3.37.2 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`.

3.37.3 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.

3.37.4 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().

3.38 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.

3.38.1 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.

3.38.2 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.

3.38.3 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.

3.39 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.

3.39.1 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`).

3.39.2 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()`.

3.39.3 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`](openff.toolkit.topology.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`).

3.39.4 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`.

3.39.5 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.

3.39.6 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.

3.40 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.

3.40.1 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.

3.40.2 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`.

3.40.3 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.

3.40.4 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.

3.40.5 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.

3.41 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.

3.41.1 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.

3.41.2 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 `ForceField\ s` are stable and conform to the SMIRNOFF specification, the pickled `ForceField\ s` that this functionality enables are not guaranteed to be compatible with future toolkit versions.

3.41.3 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.

3.41.4 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.

3.41.5 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.

3.42 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.

3.42.1 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.

3.42.2 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.

3.42.3 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.

3.43 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.

3.43.1 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`.

3.43.2 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.

3.43.3 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.

3.44 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.

3.44.1 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.

3.44.2 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.

3.44.3 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.

3.44.4 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.

3.45 0.3.0 - API Improvements

Several improvements and changes to public API.

3.45.1 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/.

3.45.2 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.

3.45.3 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

3.46 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.

3.46.1 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

3.47 0.2.1 - Bugfix release

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

3.47.1 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

3.48 0.2.0 - Initial RDKit support

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

3.48.1 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

3.49 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].

3.49.1 New features

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

3.49.2 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 I'm having troubles installing the OpenFF Toolkit on my Apple Silicon Mac.

As of August 2022, some upstreams (at least AmberTools, possibly more) are not built on `osx-arm64`, so installing the OpenFF stack is only possible with Rosetta. See the platform support section of the installation documentation for more.

(Keywords `osx-arm64`, M1 Mac, M2 Mac)

4.9 My mamba/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 environment, explicitly passing the channels to be used, in-order:

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

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

4.10 My mamba/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.

4.11 How are partial charges assigned in a SMIRNOFF force field?

There are many charge methods supported by the SMIRNOFF specification. With the exception of water, mainline OpenFF force fields only use AM1-BCC (through `ToolkitAM1BCC`) to assign partial charges. (A future biopolymer force field will likely use library charges for standard residues.)

If OpenEye Toolkits are installed and licensed, the ELF10 variant of AM1-BCC is used. OpenEye's Quacpac (`oequacpac.OEAM1BCCELF10Charges`) is used to generate partial charges.

Otherwise, RDKit is used to generate a conformer which is passed to AmberTool's `sqm` (with `-c bcc`).

Note that, because of differences with the ELF10 variant and other subtle differences between OpenEye Toolkits and RDKit/AmberTools, **assigned partial charges can be expected to differ** based on the available toolkit(s). These numerical differences are often minor but in some molecules or use cases can be significant.

A future charge method may use NAGL to assign partial charges from a graph-convolutional neural network instead of an underlying semi-empirical method. This approach is anticipated to be faster, more scalable, and more consistent than current approaches. As of March 2024, this is under development and not released for general use.

4.12 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()*.

4.13 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.

4.14 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 via a conda (or PyPI) package:

```
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.

4.15 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.

4.16 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 an `Interchange` for painless conversion to other 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.

```
from openff.toolkit import Molecule, Topology
```

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)

# First index is the Nitrogen
assert mapped_smiles.atoms[0].atomic_number == 7

# Final indices are all H
assert all([a.atomic_number == 1 for a in mapped_smiles.atoms[6:]])

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:

```
from openff.toolkit.utils.exceptions import UndefinedStereochemistryError

try:
    smiles_non_isomeric = Molecule.from_smiles("CC([NH3+])C(=O)[O-]")
except UndefinedStereochemistryError as error:
    print(error)
```

```
Unable to make OFFMol from OEMol: OEMol has unspecified stereochemistry. oemol.GetTitle()='
↪'Problematic atoms are:
```

(continues on next page)

(continued from previous page)

```
Atom atomic num: 6, name: , idx: 1, aromatic: False, chiral: True with bonds:
bond order: 1, chiral: False to atom atomic num: 6, name: , idx: 0, aromatic: False, chiral:␣
↪False
bond order: 1, chiral: False to atom atomic num: 7, name: , idx: 2, aromatic: False, chiral:␣
↪False
bond order: 1, chiral: False to atom atomic num: 6, name: , idx: 3, aromatic: False, chiral:␣
↪False
bond order: 1, chiral: False to atom atomic num: 1, name: , idx: 9, aromatic: False, chiral:␣
↪False
```

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")
```

(continues on next page)

(continued from previous page)

```
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
)
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,
```

(continues on next page)

(continued from previous page)

```
            "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,
            "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 PDB file

The `Topology.from_pdb()` method is now the recommended method for loading all PDB files. It can interpret proteins, waters, ions, and small molecules from a PDB file as a Topology. 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 input PDB file. The following block loads a PDB file containing a single protein.

```
from openff.toolkit.utils import get_data_file_path

path = get_data_file_path("proteins/T4-protein.pdb")
topology = Topology.from_pdb(path)
protein = topology.molecule(0)
protein.visualize("nglview")
```

```

```

```
NGLWidget()
```

The following block loads a PDB containing two small molecules from PDB and SMILES.

```
top_from_pdb_from_smiles = Topology.from_pdb(
    get_data_file_path("molecules/po4_phenylphosphate.pdb"),
    unique_molecules=[
        Molecule.from_smiles("P(=O)([O-])([O-])([O-])"),
        Molecule.from_smiles("C1=CC=CC=C1OP(=O)([O-1])([O-1])"),
    ],
)

top_from_pdb_from_smiles.molecule(0).visualize("nglview")
```

```
NGLWidget()
```

```
top_from_pdb_from_smiles.molecule(1).visualize("nglview")
```

```
NGLWidget()
```

And for a maximalist example, the following block loads a single PDB file containing a protein, waters, ions, and a small molecule (the chemical identity of any small molecules must be provided with the `unique_molecules` keyword argument, and any small molecules' connectivity must have corresponding CONECT records in the PDB file).

```
from openff.toolkit.utils import get_data_file_path

ligand_path = get_data_file_path("molecules/PT2385.sdf")
ligand = Molecule.from_file(ligand_path)

complex_path = get_data_file_path("proteins/5tbm_complex_solv_box.pdb")
topology = Topology.from_pdb(complex_path, unique_molecules=[ligand])

molecule_smileses = [mol.to_smiles() for mol in topology.molecules]
counts = molecule_smileses
for smiles in sorted(set(molecule_smileses)):
    count = molecule_smileses.count(smiles)
    if len(smiles) > 1000:
        smiles = "protein"
    print(smiles, ":", count, "molecule(s)")
```

```
[Cl-] : 12 molecule(s)
[H]O[H] : 4413 molecule(s)
protein : 1 molecule(s)
[H]c1c(c(c2c(c1Oc3c(c(c(c(c3[H])F)[H])C
↪#N)[H])C(C([C@@]2([H])O[H])(F)F)([H])[H])S(=O)(=O)C([H])([H])[H])[H] : 1 molecule(s)
[Na+] : 17 molecule(s)
```

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()
```

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

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,  # Note these two mappings,
        4: 3,  # which are flipped!
        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: [H:3][C@@:2]([C:5](=[O:6])[O-
↪:7])([C:1]([H:8])([H:9])[H:10])[N+:4]([H:11])([H:12])[H:13]
After remapping:  [H:3][C@@:2]([C:4](=[O:6])[O-
↪:7])([C:1]([H:8])([H:9])[H:10])[N+:5]([H:11])([H:12])[H:13]
```

```
<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 can also be used to create an OpenFF `Topology` from an MDTraj `Topology`. This method has similar restrictions and behavior as `Topology.from_openmm`.

```
from mdtraj import load_pdb

mdtraj_topology = load_pdb("zw_l_alanine.pdb").topology
openff_topology = Topology.from_mdtraj(mdtraj_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)

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.

```
client = qcportal.PortalClient("https://api.qcarchive.molssi.org:443/")

record = [*client.query_molecules(molecular_formula="C16H20N3O5")][-1]

from_qcarchive = Molecule.from_qcschema(record)

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_dataset(
    dataset_type="optimization",
    dataset_name="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

COOKBOOK: USING PDB FILES WITH THE OPENFF TOOLKIT

PDB files are a common way to represent biopolymer structures, but they don't natively contain all of the chemical information required to construct `Molecule` objects. However, PDB files are used widely in the molecular simulation community, so OpenFF has implemented functionality to load them.

The recommended pathway for loading PDB files is `Topology.from_pdb`. Additionally, the PDB file must have the following characteristics:

For protein atoms (`ATOM` records):

- Atom names and residue names must be consistent with the Chemical Components Dictonary
- Only the 20 canonical amino acids are supported by default (including protonated/deprotonated variants)
- All hydrogens must be explicit

For small molecules, waters, and ions (`HETATM` records):

- The elemental symbol of each atom must be identified according to the PDB spec in text columns 77 and 78
- Bonds must be identified in the CONECT records at the bottom of the file
- All unique small molecules must be identified in the `unique_molecules` keyword argument

Information

OpenFF `Topology` and `Molecule` objects store much more information than is in the PDB files they might be generated from. Don't be surprised if this Python code takes a few more seconds to load a PDB file than other tools.

7.1 PDB file with only a (single) protein

`6hvi_prepared.pdb` is protein from the Merck free energy perturbation study. It was prepared for simulation using commercial tools by Schrodinger, and doesn't have any bulk water, ions, small molecules, or a box.

```
protein = Topology.from_pdb("6hvi_prepared.pdb")

protein.visualize()
```

```
NGLWidget()
```

From here, you might use Packmol or PDBFixer to add water or other solvents without leaving Python (or there are tools available for water/solvent packing in AMBER, GROMACS, and many commercial packages). An example of using PDBFixer can be found in our "Toolkit Showcase" example.

7.2 PDB file with a protein, waters, and ions

This PDB file comes from the AMBER tutorial on making solvated simulations. We have slightly modified the procedure to create a file suitable for loading into the OpenFF Toolkit:

- TIP3P water is used instead of OPC (a four-site model)
- a cubic solvent box is used instead of an octohedron
- the N terminus of the protein is made neutral

```
ramp1 = Topology.from_pdb("RAMP1_solv_box_ions.pdb")

ramp1.visualize()
```

```
NGLWidget()
```

7.3 PDB file with one or more ligands

The following PDB is taken from the GROMACS protein-ligand tutorial (with manually-added CONECT records) and contains a protein, water, and a small molecule (ligand).

While the chemistry of water, ions, and proteins can be deduced from known templates (this is what happened under the hood in the previous two examples), small molecules are trickier. The number of amino acids and ions is relatively small, but the number of possible small molecules is tremendous so storing them in a database is not feasible. We instead ask the user to fill in the missing information that would be looked up from the CCD or hard-coded heuristics, specifically the bond order and stereochemistry. This information is stored in a `Molecule` object(s), which we ask the user to provide via the `unique_molecules` argument.

If you try to load a PDB file containing a small molecule, without providing the chemistry of the small molecule, you'll get an error message identifying the atoms that couldn't be loaded:

```
Topology.from_pdb("gromacs_solv_complex.pdb")
```

```
UnassignedChemistryInPDBError: Some bonds or atoms in the input could not be identified.

Hint: The following residue names with unassigned atoms were not found in the substructure␣
↪library. While the OpenFF Toolkit identifies residues by matching chemical substructures␣
↪rather than by residue name, it currently only supports the 20 'canonical' amino acids.
    JZ4


Hint: The following residues were assigned names that do not match the residue name in the␣
↪input, or could not be assigned residue names at all. This may indicate that atoms are␣
↪missing from the input or some other error. The OpenFF Toolkit requires all atoms,␣
```

(continues on next page)

(continued from previous page)

```
↪including hydrogens, to be explicit in the input to avoid ambiguities in protonation state␣
↪or bond order:
    Input residue X:JZ4#0001 contains atoms matching substructures {'No match'}

Error: The following 22 atoms exist in the input but could not be assigned chemical␣
↪information from the substructure library:
    Atom  2614 (C4) in residue X:JZ4#0001
    Atom  2615 (C7) in residue X:JZ4#0001
    Atom  2616 (C8) in residue X:JZ4#0001
    Atom  2617 (C9) in residue X:JZ4#0001
    Atom  2618 (C10) in residue X:JZ4#0001
    Atom  2619 (C11) in residue X:JZ4#0001
    Atom  2620 (C12) in residue X:JZ4#0001
    Atom  2621 (C13) in residue X:JZ4#0001
    Atom  2622 (C14) in residue X:JZ4#0001
    Atom  2623 (OAB) in residue X:JZ4#0001
    Atom  2624 (H1) in residue X:JZ4#0001
    Atom  2625 (H2) in residue X:JZ4#0001
    Atom  2626 (H3) in residue X:JZ4#0001
    Atom  2627 (H4) in residue X:JZ4#0001
    Atom  2628 (H5) in residue X:JZ4#0001
    Atom  2629 (H6) in residue X:JZ4#0001
    Atom  2630 (H7) in residue X:JZ4#0001
    Atom  2631 (H8) in residue X:JZ4#0001
    Atom  2632 (H9) in residue X:JZ4#0001
    Atom  2633 (H10) in residue X:JZ4#0001
    Atom  2634 (H11) in residue X:JZ4#0001
    Atom  2635 (H12) in residue X:JZ4#0001
```

Any OpenFF `Molecule` object with the appropriate atoms and connectivity can be used to identify the small molecule when loading the PDB file. See the Molecule Cookbook for some of the numerous ways to create them! Since the coordinates are specified in the PDB file, it's not necessary that this `Molecule` has conformers.

In this case we know the identity of the ligand and can look up a SMILES string, which is all that's needed to load this PDB file! Since there are many molecules in this file, it may take a few tens of seconds to load and visualize - this is expected and is a result of carefully checking and applying chemical information.

```
jz4 = Molecule.from_smiles("CCCC1=CC=CC=C1O")
complex = Topology.from_pdb("gromacs_solv_complex.pdb", unique_molecules=[jz4])

complex.visualize()
```

```
NGLWidget()
```

CHAPTER

EIGHT

SMIRNOFF (SMIRKS NATIVE OPEN FORCE FIELD)

8.1 The SMIRNOFF specification

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

8.2 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

NINE

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'.

9.1 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 ± 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.

9.2 Ordering of atoms and virtual sites

The OpenFF Toolkit and Interchange currently add all new virtual particles to the "end" of a Topology, such that the particle indices of all newly-created virtual particles are higher than index of the last atom.

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

TEN

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.

10.1 Overview

10.1.1 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.

10.1.2 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 `main` 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 `main` 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 `main` 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.

10.1.3 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.

10.2 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.

10.2.1 `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.

10.2.2 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.

10.2.3 `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`.

10.2.4 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.

10.3 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.

10.3.1 Setting up a development environment

1. Install the `mamba` package manager as part of the Miniforge or Mambaforge distributions (an alternative to the Anaconda Distribution)
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
mamba env create -n openff-dev -f devtools/conda-envs/test_env.yaml
mamba activate openff-dev
```

(continues on next page)

(continued from previous page)

```
# 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
mamba env update --name openff-dev --file docs/environment.yml
mamba install --name openff-dev -c conda-forge sphinx-autobuild
```

To build the documentation from scratch:

```
# Build the documentation
# From the openff-toolkit root directory
mamba 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
mamba activate openff-dev
sphinx-autobuild docs docs/_build/html --watch openff
# Then navigate your web browser to http://localhost:8000
```

10.3.2 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:

```
mamba install black isort flake8 -c conda-forge
black openff
isort openff
flake8 openff
```

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

10.3.3 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:

```
mamba 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.

You can also manually run all hooks outside of commit time with

```
pre-commit run
```

or an individual hook by name, i.e.

```
pre-commit run flake8
```

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.

10.3.4 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.

10.3.5 Jupyter Notebooks in Rendered Docs

We sometimes include Jupyter Notebooks in our book-style documentation. These are files with the `.ipynb` extension. Notebooks allow sections of the docs that include a large number of code examples to be tested easily, and support automatically rendering example output, including interactive Jupyter widgets such as the 3D molecule representations in the *molecule cookbook*. Note that the following information only applies to notebooks in the `docs` directory; notebooks in the `examples` directory are not included in the Toolkit docs (but may be included in OpenFF Examples via a different process).

Information

For interactive widgets to appear in rendered documentation, they must be stored in the notebook each time it is re-executed. For information on how to do this, see *Widgets*.

MyST Markdown

Documentation notebooks are rendered with the MyST-NB Sphinx extension, which provides the expanded Markdown syntax used in other Markdown files in this documentation. Admonitions, maths, and links to API references and other documentation pages can be included in the usual ways. This means that some correct Markdown will appear incorrect in Jupyter by default, as it uses a simpler Markdown dialect. To render MyST syntax, you can install the `jupyterlab-myst` Jupyter extension:

```
mamba install -c conda-forge jupyterlab-myst
```

Note that some features that will render correctly in Sphinx rely on extensions that are not available to `jupyterlab-myst`, and so may still appear incorrect even with the extension.

Widgets

Jupyter Notebooks can produce interactive widgets as output, and MyST-NB supports rendering them in HTML output. Any given widget includes both Python code and JavaScript code, and Jupyter defines ways for them to speak to each other which allows Python code to be run interactively via the widget in the notebook.

While the JavaScript components are embedded in the page by Sphinx and can run in the browser's JavaScript engine, Sphinx web pages do not include a Python runtime. This has two effects. Firstly, any interactivity that relies on the Python runtime will not work on the web. This is usually little more than an inconvenience for purpose-written documentation notebooks, but does explain why NGLView trajectories in OpenFF Examples cannot be played on the web.

Secondly, widget initialization happens in Python, so any state shared between Python and JavaScript must be included in the notebook itself before Sphinx can render it. Any time these notebooks are re-executed, the internal IDs of the widgets may change, causing the widget to forget its original state and be missing from the rendered documentation. To fix this, it's essential to ask Jupyter to **save widget state**. In Jupyter Notebook 7+ and Jupyter Lab, this option appears under `Settings -> Save Widget State Automatically`, which if ticked will save widget state when the notebook is saved. Note that this menu item will not appear if the open notebook has no widgets. In earlier versions of Notebook, the `Widgets -> Save Widget State` menu item must be selected any time the notebook is re-executed and saved.

Widget state can also be saved programmatically by executing the notebook with NBConvert, though since NBConvert does not include a JavaScript runtime the widget must be able to initialize itself from stored state. This is not possible, for example, in NGLView before version 3.0.6.

10.4 Supported Python versions

The OpenFF Toolkit roughly follows NEP 29. As of April 2023 this means Python 3.9-3.10 are officially supported (3.11 is missing some upstream dependencies). 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 3.8 and older	No support after April 2023					
Python 3.9	**Test**	**Test**	**Test**	**Test**	**Test**	**Test**
Python 3.10	**Test**	Skip	Skip	**Test**	Skip	Skip
Python 3.11	**Test**	Skip	Skip	**Test**	Skip	Skip
Python 3.12 and newer	Pending official releases and upstream support					

CHAPTER

ELEVEN

MOLECULE CONVERSION TO OTHER PACKAGES

Molecule conversion spec

11.1 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	insertion_code	chain_id
PDB file ATOM/HETATM columns	Columns 18-20 (resName)	Columns 23-26 (resSeq)	Columns 27 (iCode)	Columns 22 (chainID)
PDBx/MMCIF fields	label_comp_id	label_seq_id	label_ins_code	label_asym_id
OpenFF getter (defined)	`atom.metadata["residue_name"]`	`atom.metadata["residue_number"]`	`atom.metadata["insertion_code"]`	`atom.metadata["chain_id"]`
OpenFF getter (undefined)	`"residue_name" not in atom.metadata`	`"residue_number" not in atom.metadata`	`"insertion_code" 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["insertion_code"] = X`	`atom.metadata["chain_id"] = X`
OpenFF setter (undefined)	`del atom.metadata['residue_name']`	`del atom.metadata['residue_number']`	`del atom.metadata['insertion_code']`	`del atom.metadata['chain_id']`
OpenMM getter (defined)	`omm_atom.residue.name`	`omm_atom.residue.id`	`omm_atom.residue.insertionCode`	`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	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.insertionCode = X`	`omm_atom.residue.chain.id = X`
OpenMM setter (undefined)	`omm_atom.residue.name = "UNL"`	`omm_atom.residue.id = 0`	`omm_atom.residue.insertionCode = " "`	`omm_atom.residue.chain.id = "X"`
RDKit getter (defined)	`rda.GetPDBResidueInf GetResidueName()`	`rda.GetPDBResidueInf GetResidueNumber`	`rda.GetPDBResidueInf GetInsertionCode`	`rda.GetPDBResidueInfo().GetChainId()`
RDKit getter (undefined)	`rda.GetPDBResidueInf is None`	`rda.GetPDBResidueInf is None`	`rda.GetPDBResidueInf is None`	`rda.GetPDBResidueInfo() is None`
RDKit setter (defined)	`res = rda.`	`res = rda.`	`res = rda.`	`.res = rda.`

CHAPTER

TWELVE

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).

12.1 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.

12.1.1 `FrozenMolecule`

class `openff.toolkit.topology.`**`FrozenMolecule`**(*other=None*, *file_format: str | None = None*, *toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY*, *allow_undefined_stereo: bool = 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: str | None = None, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY, allow_undefined_stereo: bool = False*)

Create a new FrozenMolecule object

Parameters

- **other** – 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** – 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 registry to use for I/O operations
- **allow_undefined_stereo** – 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 = FrozenMolecule(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
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 protomers.
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 a Molecule from a SMILES string, ordering atoms from mappings
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_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[, ...])	Loads a polymer from a PDB file.
from_qcschema(qca_object[, ...])	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

continues on next page

Table 1 – continued from previous page

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 from the element symbol and count.
get_available_charge_methods([toolkit_registry]	Get the charge methods supported by each wrapper in the specified registry.
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
remap(mapping_dict[, current_to_new, partial])	Reorder the atoms in the molecule according to 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.
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.

continues on next page

Table 1 – continued from previous page

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_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.
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 from the element symbol and count.

Names are generated from the elemental symbol and the number of times that element is found in the molecule. 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. For example, generated atom names might begin 'C1x', 'H1x', 'O1x', 'C2x', etc.

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

to_dict() → dict

Return a dictionary representation of the molecule.

Returns
molecule_dict – A dictionary representation of the molecule.

ordered_connection_table_hash() → int

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

classmethod from_dict(*molecule_dict: dict*) → FM

Create a new Molecule from a dictionary representation

Parameters
molecule_dict – A dictionary representation of the molecule.

Returns
molecule – A Molecule created from the dictionary representation

add_default_hierarchy_schemes(*overwrite_existing: bool = 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 – 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.

See also:

HierarchyScheme, Molecule.add_hierarchy_scheme, Molecule.update_hierarchy_schemes, Molecule.perceive_residues

add_hierarchy_scheme(*uniqueness_criteria: Iterable[str], iterator_name: str*) → *HierarchyScheme*

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** – 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** – Name of the iterator that will be exposed to access the hierarchy elements generated by this scheme. Must not match an existing attribute of the `Molecule`, i.e. `atoms`, `angles`, etc.

Returns

new_hier_scheme – The newly created HierarchyScheme

See also:

`Molecule.add_default_hierarchy_schemes`, `Molecule.hierarchy_schemes`, `Molecule.delete_hierarchy_scheme`, `Molecule.update_hierarchy_schemes`, `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.

See also:

`Molecule.add_hierarchy_scheme`, `Molecule.delete_hierarchy_scheme`, `Molecule.update_hierarchy_schemes`, `Topology.hierarchy_iterator`, `HierarchyScheme`

delete_hierarchy_scheme(*iter_name: str*)

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 –

See also:

`Molecule.add_hierarchy_scheme`, `Molecule.update_hierarchy_schemes`, `Molecule.hierarchy_schemes`, `HierarchyScheme`

update_hierarchy_schemes(*iter_names: list[str] | None = 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 – Only perceive hierarchy for HierarchySchemes that expose these iter-

ator names. If not provided, all known hierarchies will be perceived, overwriting previous results if applicable.

See also:

`Molecule.add_hierarchy_scheme`, `Molecule.delete_hierarchy_schemes`, `Molecule.hierarchy_schemes`, `HierarchyScheme`

to_smiles(*isomeric: bool = True, explicit_hydrogens: bool = True, mapped: bool = False, toolkit_registry: ToolkitRegistry | ToolkitWrapper = 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** – return an isomeric smiles
- **explicit_hydrogens** – return a smiles string containing all hydrogens explicitly
- **mapped** – 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** – `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES conversion

Returns

smiles – 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: str, allow_undefined_stereo: bool = False, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY, name: str = ''*) → FM

Construct a Molecule from a InChI representation

Parameters

- **inchi** – The InChI representation of the molecule.
- **allow_undefined_stereo** – 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** – `ToolkitRegistry` or `ToolkitWrapper` to use for InChI-to-molecule conversion

- **name** – An optional name for the output molecule

Returns
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: bool = False, toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY*) → str

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** – 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** – `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: bool = False, toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *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** – 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** – `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: str, hydrogens_are_explicit: bool = False, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY, allow_undefined_stereo: bool = False, name: str = ''*) → FM

Construct a `Molecule` from a SMILES representation

The order of atoms in the `Molecule` is unspecified and may change from version to version or with different toolkits. SMILES atom indices (also known as atom maps) are not used to order atoms; instead, they are stored in the produced molecule's properties attribute, accessible via `molecule.properties["atom_map"]`. The atom map is stored as a dictionary mapping molecule atom indices to SMILES atom maps. To order atoms according to SMILES atom indices, see `Molecule.from_mapped_smiles()`, which helpfully raises an exception if any atom map is missing, duplicated, or out-of-range, or else `Molecule.remap()` for arbitrary remaps.

Parameters

- **smiles** – The SMILES representation of the molecule.
- **hydrogens_are_explicit** – If `True`, forbid the cheminformatics toolkit from inferring hydrogen atoms not explicitly specified in the SMILES.
- **toolkit_registry** – The cheminformatics toolkit to use to interpret the SMILES.
- **allow_undefined_stereo** – 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.
- **name** – An optional name for the output molecule

Raises

RadicalsNotSupportedError – If any atoms in the input molecule contain radical electrons.

Examples

Create a `Molecule` representing toluene from SMILES:

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

Create a `Molecule` representing phenol from SMILES with the oxygen at atom index 0 (SMILES indices begin at 1):

```
>>> molecule = Molecule.from_smiles('c1ccccc1[OH:1]')
>>> molecule = molecule.remap(
...     {k: v - 1 for k, v in molecule.properties["atom_map"].items()},
...     partial=True,
... )
>>> assert molecule.atom(0).symbol == "O"
```

See also:

`from_mapped_smiles`, `remap`

static are_isomorphic(*mol1:* FrozenMolecule | *_SimpleMolecule* | *Graph*, *mol2:* FrozenMolecule | *_SimpleMolecule* | *Graph*, *return_atom_map:* *bool* = *False*, *aromatic_matching:* *bool* = *True*, *formal_charge_matching:* *bool* = *True*, *bond_order_matching:* *bool* = *True*, *atom_stereochemistry_matching:* *bool* = *True*, *bond_stereochemistry_matching:* *bool* = *True*, *strip_pyrimidal_n_atom_stereo:* *bool* = *True*, *toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY*) → tuple[bool, Optional[dict[int, int]]]

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** – The first molecule to test for isomorphism.
- **mol2** – The second molecule to test for isomorphism.
- **return_atom_map** – Return a `dict` containing the atomic mapping, otherwise `None`. Only processed if inputs are isomorphic, will always return `None` if inputs are not isomorphic.
- **aromatic_matching** – If `False`, aromaticity of graph nodes and edges are ignored for the purpose of determining isomorphism.
- **formal_charge_matching** – If `False`, formal charges of graph nodes are ignored for the purpose of determining isomorphism.
- **bond_order_matching** – If `False`, bond orders of graph edges are ignored for the purpose of determining isomorphism.
- **atom_stereochemistry_matching** – If `False`, atoms' stereochemistry is ignored for the purpose of determining isomorphism.
- **bond_stereochemistry_matching** – If `False`, bonds' stereochemistry is ignored for the purpose of determining isomorphism.

- **strip_pyrimidal_n_atom_stereo** – If `True`, any stereochemistry defined around pyrimidal nitrogen stereocenters will be disregarded in the isomorphism check.
- **toolkit_registry** – `ToolkitRegistry` or `ToolkitWrapper` to use for removing stereochemistry from pyrimidal nitrogens.

Returns

- *molecules_are_isomorphic*
- *atom_map* – [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:* FrozenMolecule | *_SimpleMolecule* | *Graph*, ***kwargs*) → bool

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** –
- **aromatic_matching** –
- **atoms.** (compare the formal charges attributes of the) –
- **formal_charge_matching** –
- **atoms.** –
- **bond_order_matching** –
- **bonds.** (compare the bond order on attributes of the) –
- **atom_stereochemistry_matching** – If `False`, atoms' stereochemistry is ignored for the purpose of determining equality.
- **bond_stereochemistry_matching** – If `False`, bonds' stereochemistry is ignored for the purpose of determining equality.
- **strip_pyrimidal_n_atom_stereo** – If `True`, any stereochemistry defined around pyrimidal nitrogen stereocenters will be disregarded in the isomorphism check.
- **toolkit_registry** – `ToolkitRegistry` or `ToolkitWrapper` to use for removing stereochemistry from pyrimidal nitrogens.

Returns

isomorphic

generate_conformers(*toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY*, *n_conformers:* *int* = *10*, *rms_cutoff:* *Quantity* | *None* = *None*, *clear_existing:* *bool* = *True*, *make_carboxylic_acids_cis:* *bool* = *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** – ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion
- **n_conformers** – The maximum number of conformers to produce
- **rms_cutoff** – 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** – Whether to overwrite existing conformers for the molecule
- **make_carboxylic_acids_cis** – 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: ToolkitRegistry | ToolkitWrapper | None = 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`

get_available_charge_methods(*toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY*) → list[str]

Get the charge methods supported by each wrapper in the specified registry.

Parameters
toolkit_registry – `ToolkitRegistry` or `ToolkitWrapper` to use for the calculation.

assign_partial_charges(*partial_charge_method: str, strict_n_conformers: bool = False, use_conformers: Iterable[Quantity] | None = None, toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY, normalize_partial_charges: bool = 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"`

By default, the conformers on the input molecule are not used in the charge calculation. Instead, any conformers needed for the charge calculation are generated by this method. If this behavior is undesired, specific conformers can be provided via the `use_conformers` argument.

ELF10 methods will neither fail nor warn when fewer than the expected number of conformers could be generated, as many small molecules are too rigid to provide a large number of conformers. Note that only the `"am1bccelf10"` partial charge method uses ELF conformer selection; the `"am1bcc"` method only uses a single conformer. This may confuse users as the ToolkitAM1BCC SMIRNOFF tag in a force field file defines that AM1BCC-ELF10 should be used if the OpenEye Toolkits are available.

For more supported charge methods and their 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** – The partial charge calculation method to use for partial charge calculation.
- **strict_n_conformers** – 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** – Coordinates to use for partial charge calculation. If `None`, an appropriate number of conformers will be generated.
- **toolkit_registry** – `ToolkitRegistry` or `ToolkitWrapper` to use for the calculation.
- **normalize_partial_charges** – 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

Generate AM1 Mulliken partial charges. Conformers for the AM1 calculation are generated automatically:

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

To use pre-generated conformations, use the `use_conformers` argument:

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.generate_conformers(n_conformers=1)
>>> molecule.assign_partial_charges(
...     'am1-mulliken',
...     use_conformers=molecule.conformers
... )
```

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: str | None = None, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY, use_conformers: Iterable[Quantity] | None = 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** – `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion
- **bond_order_model** – The bond order model to use for fractional bond order calculation. If `None`, `"am1-wiberg"` is used.
- **use_conformers** – The conformers to use for fractional bond order calculation. If `None`, an appropriate number of conformers will be generated by an available `ToolkitWrapper`.

Examples

```
>>> from openff.toolkit import Molecule
>>> 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() → Graph

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 – 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: list[str] | None = None, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY*) → list[*Bond*]

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

Parameters

- **ignore_functional_groups** – A list of bond SMARTS patterns to be ignored when finding rotatable bonds.
- **toolkit_registry** – `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 – The partial charges on the molecule's atoms. Returns None if no charges have been specified.

property n_atoms: int

The number of Atom objects.

property n_bonds: int

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 atoms

Iterate over all Atom objects in the molecule.

atom(*index: int*) → *Atom*

Get the atom with the specified index.

Parameters
index –

Returns
atom

atom_index(*atom:* Atom) → int

Returns the index of the given atom in this molecule

Parameters
atom –

Returns
index – 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 –

Returns
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

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 – 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 – 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 – 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 – 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.

chemical_environment_matches(*query: str*, *unique: bool = False*, *toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY*)

Find matches in the molecule for a SMARTS string

Parameters

- **query** – SMARTS string (with one or more tagged atoms).
- **unique** – If `True`, de-duplicates matches before returning.
- **toolkit_registry** – `ToolkitRegistry` or `ToolkitWrapper` to use for chemical environment matches

Returns
matches – 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: str, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY, allow_undefined_stereo: bool = False, **kwargs*) → FM

Generate a molecule from IUPAC or common name

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

Parameters

- **iupac_name** – IUPAC name of molecule to be generated
- **toolkit_registry** – `ToolkitRegistry` or `ToolkitWrapper` to use for chemical environment matches
- **allow_undefined_stereo** – If false, raises an exception if molecule contains undefined stereochemistry.

Returns
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* – 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*) → FM

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

Parameters

topology – The `Topology` object containing a single `Molecule` object. Note that OpenMM and MDTraj `Topology` objects are not supported.

Returns

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

```
>>> from openff.toolkit import Molecule, Topology
>>> topology = Topology.from_molecules(Molecule.from_smiles('[CH4]'))
>>> molecule = Molecule.from_topology(topology)
```

to_topology()

Return an OpenFF Topology representation containing one copy of this molecule

Returns

topology – A Topology representation of this molecule

Examples

```
>>> from openff.toolkit import Molecule
>>> molecule = Molecule.from_iupac('imatinib')
>>> topology = molecule.to_topology()
```

classmethod from_file(*file_path: str | Path | TextIO, file_format=None, toolkit_registry=GLOBAL_TOOLKIT_REGISTRY, allow_undefined_stereo: bool = False*) → FM | list[FM]

Create one or more molecules from a file

Parameters

- **file_path** – The path to the file or file-like object to stream one or more molecules from.
- **file_format** – 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** – `ToolkitRegistry` or `ToolkitWrapper` to use for file loading. If a Toolkit is passed, only the highest-precedence toolkit is used

- **allow_undefined_stereo** – If false, raises an exception if oemol contains undefined stereochemistry.

Returns
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 import Molecule
>>> from openff.toolkit.utils.utils import get_data_file_path
>>> sdf_file_path = get_data_file_path("molecules/toluene.sdf")
>>> molecule = Molecule.from_file(sdf_file_path)
```

classmethod from_polymer_pdb(*file_path: str | Path | TextIO*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *name: str = ''*) → FM

Loads a polymer from a PDB file.

Also see `Topology.from_multicomponent_pdb()`, which can do everything this method can and more.

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. Only one polymer chain may be present in the PDB file, and it must be the only molecule present.

Connectivity and bond orders are assigned by matching SMARTS codes for the supported residues against atom names. The PDB file must include all atoms with the correct standard atom names described in the PDB Chemical Component Dictionary. Residue names are used to assist troubleshooting failed assignments, but are not used in the actual assignment process.

Metadata such as residues, chains, and atom names are recorded in the `Atom.metadata` attribute, which is a dictionary mapping from strings like "residue_name" 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`.

Parameters

- **file_path** – PDB information to be passed to OpenMM PDBFile object for loading
- **None** (`toolkit_registry = ToolkitWrapper or ToolkitRegistry. Default =`) – Either a ToolkitRegistry, ToolkitWrapper
- **name** – An optional name for the output molecule

Returns
molecule

Raises

- **UnassignedChemistryInPDBError** – If an atom or bond could not be assigned; the exception will provide a detailed diagnostic of what went wrong.
- **MultipleMoleculesInPDBError** – If all atoms and bonds could be assigned, but the PDB includes multiple chains or molecules.

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** – A file-like object or the path to the file to be written.
- **file_format** – Format specifier, one of ['MOL2', 'MOL2H', 'SDF', 'PDB', 'SMI', 'CAN', 'TDT'] Note that not all toolkits support all formats
- **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** – The maximum amount of molecules that should be returned
- **toolkit_registry** – `ToolkitRegistry` or `ToolkitWrapper` to use to enumerate the tautomers.

Returns

molecules – A list of openff.toolkit.topology.Molecule instances not including the input molecule.

enumerate_stereoisomers(*undefined_only: bool = False, max_isomers: int = 20, rationalise: bool = True, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY*)

Enumerate the stereocenters and bonds of the current molecule.

Parameters

- **undefined_only** – If we should enumerate all stereocenters and bonds or only those with undefined stereochemistry
- **max_isomers** – The maximum amount of molecules that should be returned
- **rationalise** – If we should try to build and rationalise the molecule to ensure it can exist
- **toolkit_registry** – `ToolkitRegistry` or `ToolkitWrapper` to use to enumerate the stereoisomers.

Returns

molecules – A list of `Molecule` instances not including the input molecule.

enumerate_protomers(*max_states: int = 0*) → list

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

Parameters

max_states – The maximum number of protomer states to be returned. If 0, the default, attempt to return all protomers. If set to a non-zero number, the input molecule is not guaranteed to be included in the returned list.

Returns

molecules – A list of the protomers of the input molecules, including the input molecule if found by the underlying toolkit's protomer enumeration tool and not pruned by *max_states*.

classmethod from_rdkit(*rdmol*, *allow_undefined_stereo: bool = False*, *hydrogens_are_explicit: bool = False*) → FM

Create a Molecule from an RDKit molecule.

Requires the RDKit to be installed.

Parameters

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

Returns

molecule – An OpenFF molecule

Examples

Create a molecule from an RDKit molecule

```
>>> from openff.toolkit import Molecule
>>> from rdkit import Chem
>>> rdmol = Chem.MolFromSmiles("CCO")
>>> molecule = Molecule.from_rdkit(rdmol)
```

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

Create an RDKit molecule

Requires the RDKit to be installed.

Parameters

aromaticity_model – The aromaticity model to use. Only OEAroModel_MDL is supported.

Returns

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: bool = False*) → *FrozenMolecule*

Create a `Molecule` from an OpenEye molecule.

Requires the OpenEye toolkit to be installed.

Parameters

- **oemol** – An OpenEye molecule
- **allow_undefined_stereo** – If `False`, raises an exception if oemol contains undefined stereochemistry.

Returns

molecule – An OpenFF molecule

Examples

Create a `Molecule` from an OpenEye OEMol

```
>>> from openff.toolkit import Molecule
>>> from openeye import oechem
>>> oemol = oechem.OEMol()
>>> oechem.OESmilesToMol(oemol, '[H]C([H])([H])C([H])([H])O[H]')
True
>>> molecule = Molecule.from_openeye(oemol)
```

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

Create a QCElemental Molecule.

The kekule structure of the molecule is saved in two places on the returned Molecule:

- `extras["canonical_isomeric_explicit_hydrogen_mapped_smiles"]`
- `identifiers["canonical_isomeric_explicit_hydrogen_mapped_smiles"]`

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

Parameters

- **multiplicity** – The multiplicity of the molecule; sets `molecular_multiplicity` field for QCElemental Molecule.
- **conformer** – The index of the conformer to use for the QCElemental Molecule geometry.
- **extras** – 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: str, toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY, allow_undefined_stereo: bool = False*) → FM

Create a `Molecule` from a SMILES string, ordering atoms from mappings

SMILES strings support mapping integer indices to each atom by ending a bracketed atom declaration with a colon followed by a 1-indexed integer:

This method creates a `Molecule` from such a SMILES string whose atoms are ordered according to the mapping. Each atom must be mapped exactly once; any duplicate, missing, or out-of-range mappings will cause the method to fail.

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

Parameters

- **mapped_smiles** (`str`) – A mapped SMILES string with explicit hydrogens.
- **toolkit_registry** – Cheminformatics toolkit to use for SMILES-to-molecule conversion
- **allow_undefined_stereo** – If false, raise an exception if the SMILES contains undefined stereochemistry.

Returns
offmol – An OpenFF molecule instance.

Raises

- **SmilesParsingError** – If the given SMILES had no indexing picked up by the toolkits, or if the indexing is missing indices.
- **RemapIndexError** – If the mapping has duplicate or out-of-range indices.

Examples

Create a mapped chlorofluoroiodomethane molecule and check the atoms are placed accordingly:

```
>>> molecule = Molecule.from_mapped_smiles(
...     "[Cl:2][C@:1]([F:3])([I:4])[H:5]"
... )
>>> assert molecule.atom(0).symbol == "C"
>>> assert molecule.atom(1).symbol == "Cl"
>>> assert molecule.atom(2).symbol == "F"
>>> assert molecule.atom(3).symbol == "I"
>>> assert molecule.atom(4).symbol == "H"
```

See also:

`from_smiles`, `remap`

classmethod from_qcschema(*qca_object*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *allow_undefined_stereo: bool = False*)

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

If this method is provided a QCElemental Molecule (or dict representation of a Molecule), it will return a single-conformer OpenFF Molecule.

If this method is provided a QCFractal dataset Entry (or dict representation of an Entry), it will return an OpenFF Molecule with at least one conformer, corresponding to the:

- `.molecule` attribute of a SinglepointDatasetEntry (single conformer)
- `.initial_molecule` attribute of an OptimizationDatasetEntry or GridoptimizationDatasetEntry (single conformer)
- `initial_molecules` attribute of a TorsiondriveDatasetEntry (one or more conformers, in the order that they appear when accessing the `initial_molecules` attribute on the Entry object)

If these QC molecules have their `.id` fields populated, the returned OpenFF Molecule will have a dict mapping QC IDs to conformer numbers (`offmol.properties["initial_molecules"]`)

The data source must also specify the kekule structure of the molecule. Currently the only supported format for this is in the `canonical_isomeric_explicit_hydrogen_mapped_smiles` field, which will be taken from the following locations, if available, in the following order of priority:

- The input's `attributes` attribute (set on QCFractal DatasetEntry objects, such as `SinglepointDatasetEntry` and `TorsiondriveDatasetEntry`)
- The input's `identifiers` attribute (set on QCSchema Molecules made after QCFractal 0.50)
- The input's `extras` attribute (the information was typically set on QCSchema Molecules as part of OpenFF's QC data submission pipeline before QCFractal 0.50)

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

Parameters

- **qca_object** – A QCArchive molecule record or dataset entry, or dict representation of either.

- **toolkit_registry** – openff.toolkit.utils.toolkits.ToolkitWrapper, optional `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion
- **allow_undefined_stereo** – If false, raises an exception if qca_object contains undefined stereochemistry.

Returns
molecule – An OpenFF molecule instance.

Examples

Get Molecule from a QCArchive molecule record:

```
>>> try:
...     from qcportal import PortalClient
... except ImportError:
...     import pytest
...     pytest.skip("This tests sometimes fails when OpenEye is installed")
>>> client = PortalClient("https://api.qcarchive.molssi.org:443/")
>>> offmol = Molecule.from_qcschema(
...     [*client.query_molecules(molecular_formula="C16H20N3O5")][-1]
... )
>>> offmol.to_hill_formula()
'C16H20N3O5'
```

Get Molecule from a QCArchive optimization entry:

```
>>> from qcportal import PortalClient
>>> client = PortalClient("https://api.qcarchive.molssi.org:443/")
>>> optimizations = client.get_dataset(
...     dataset_type="optimization",
...     dataset_name="SMIRNOFF Coverage Set 1",
... )
>>> offmol = Molecule.from_qcschema(optimizations.get_entry('coc(o)oc-0'))
>>> offmol.to_hill_formula()
'C3H8O3'
```

Raises

- **InvalidQCInputError** – If the input record isn't suitable to be made into an OpenFF Molecule
- **MissingCMILESError** – If the record does not contain the `canonical_isomeric_explicit_hydrogen_mapped_smiles`.
- **InvalidConformerError** – If the conformer could not be attached.

classmethod from_pdb_and_smiles(*file_path*, *smiles*, *allow_undefined_stereo: bool = False*, *name: str = ''*) → FM

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** – PDB file path
- **smiles** – a valid smiles string for the pdb, used for stereochemistry, formal charges, and bond order
- **allow_undefined_stereo** – If false, raises an exception if SMILES contains undefined stereochemistry.
- **name** – An optional name for the output molecule

Returns
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.ToolkitWrapper, optional `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

Returns
molecule – An new OpenFF style molecule with atoms in the canonical order.

remap(*mapping_dict: dict[int, int], current_to_new: bool = True, partial: bool = False*)

Reorder the atoms in the molecule according to the given mapping dict.

The mapping dict must be a dictionary mapping atom indices to atom indices. Each atom index must be an integer in the half-open interval `[0, n_atoms)`; ie, it must be a valid index into the `self.atoms` list. All atom indices in the molecule must be mapped from and to exactly once unless `partial=True` is given, in which case they must be mapped no more than once. Missing (unless `partial=True`), out-of-range (including non-integer), or duplicate indices are not allowed in the `mapping_dict` and will lead to an exception.

By default, the mapping dict's keys are the source indices and its values are destination indices, but this can be changed with the `current_to_new` argument.

The keys of the `self.properties["atom_map"]` property are updated for the new ordering. Other values of the properties dictionary are transferred unchanged.

Warning: This API is experimental and subject to change.

Parameters

- **mapping_dict** – A dictionary of the mapping between indices. The mapping should be indexed starting from 0 for both the source and destination; note that SMILES atom mapping is typically 1-based.
- **current_to_new** – 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}`.
- **partial** – If `False` (the default), an exception will be raised if any atom is lacking a destination in the atom map. Note that if this is `True`, atoms without entries in the mapping dict may be moved in addition to those in the dictionary. Note that partial maps must still be in-range and not include duplicates.

Returns

new_molecule – A copy of the molecule in the new order.

Raises

RemapIndexError – When an out-of-range, duplicate, or missing index is found in the `mapping_dict`.

See also:

`from_mapped_smiles`

to_openeye(*toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY*, *aromaticity_model:* *str* = *DEFAULT_AROMATICITY_MODEL*)

Create an OpenEye molecule

Requires the OpenEye toolkit to be installed.

Parameters

aromaticity_model – The aromaticity model to use. Only OEAroModel_MDL is supported.

Returns

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:* *int* | Atom, *j:* *int* | Atom) → *Bond*

Returns the bond between two atoms

Parameters

- **i** – Atoms or atom indices to check
- **j** – Atoms or atom indices to check

Returns

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 – A BSON serialized representation of the object

Returns
instance – An instantiated object

classmethod from_json(*serialized: str*)

Instantiate an object from a JSON serialized representation.

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

Parameters
serialized – A JSON serialized representation of the object

Returns
instance – An instantiated object

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

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

Parameters
serialized – A MessagePack-encoded bytes serialized representation

Returns
instance – 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 – A pickled representation of the object

Returns
instance – An instantiated object

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

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

Parameters
serlialized – A TOML serialized representation of the object

Returns
instance – An instantiated object

classmethod from_xml(*serialized*)

Instantiate an object from an XML serialized representation.

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

Parameters
serialized – An XML serialized representation

Returns
instance – Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters
serialized – A YAML serialized representation of the object

Returns
instance – Instantiated object

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns
serialized – A BSON serialized representation of the objecft

to_json(*indent=None*) → str

Return a JSON serialized representation.

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

Parameters
indent – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – A JSON serialized representation of the object

to_messagepack()

Return a MessagePack representation.

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

Returns
serialized – 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 – A pickled representation of the object

to_toml()

Return a TOML serialized representation.

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

Returns
serialized – A TOML serialized representation of the object

to_xml(*indent=2*)

Return an XML representation.

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

Parameters

indent – If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized – A MessagePack-encoded bytes serialized representation.

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized – A YAML serialized representation of the object

12.1.2 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*)

See FrozenMolecule.__init__

Methods

`__init__`(*args, **kwargs)	See FrozenMolecule.__init__
`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.
`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
`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 protomers.
`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

continues on next page

Table 2 – continued from previous page

from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_mapped_smiles(mapped_smiles[, ...])	Create a Molecule from a SMILES string, ordering atoms from mappings
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_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[, ...])	Loads a polymer from a PDB file.
from_qcschema(qca_object[, ...])	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 from the element symbol and count.
get_available_charge_methods([toolkit_registry	Get the charge methods supported by each wrapper in the specified registry.
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
perceive_residues([substructure_file_path, ...])	Perceive a polymer's residues and permit iterating over them.
remap(mapping_dict[, current_to_new, partial])	Reorder the atoms in the molecule according to 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.

continues on next page

Table 2 – continued from previous page

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.
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()	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_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.
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: int, formal_charge: int, is_aromatic: bool, stereochemistry: str | None = None, name: str | None = None, metadata: dict[str, Union[int, str]] | None = None*) → int

Add an atom to the molecule.

Parameters

- **atomic_number** – Atomic number of the atom
- **formal_charge** – Formal charge of the atom
- **is_aromatic** – If True, atom is aromatic; if False, not aromatic
- **stereochemistry** – Either 'R' or 'S' for specified stereochemistry, or None if stereochemistry is irrelevant
- **name** – An optional name for the atom
- **metadata** – 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 – 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, 1, False)
>>> bond_idx = molecule.add_bond(C, H2, 1, False)
>>> bond_idx = molecule.add_bond(C, H3, 1, False)
>>> bond_idx = molecule.add_bond(C, H4, 1, False)
>>> molecule.to_smiles(explicit_hydrogens=False)
'C'
```

add_bond(*atom1: int | Atom, atom2: int | Atom, bond_order: int, is_aromatic: bool, stereochemistry: str | None = None, fractional_bond_order: float | None = None*) → int

Add a bond between two specified atom indices

Parameters

- **atom1** – Index of first atom
- **atom2** – Index of second atom
- **bond_order** – Integral bond order of Kekulized form
- **is_aromatic** – True if this bond is aromatic, False otherwise
- **stereochemistry** – Either `'E'` or `'Z'` for specified stereochemistry, or `None` if stereochemistry is irrelevant
- **fractional_bond_order** – The fractional (eg. Wiberg) bond order

Returns

index (*int*) – Index of the bond in this molecule

Examples

For an example of use, see `add_atom()`.

add_conformer(*coordinates: Quantity*) → int

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 – The index of this conformer

visualize(*backend: Literal['rdkit']*) → IPython.display.SVG

visualize(*backend: Literal['openeye']*) → IPython.display.Image

visualize(*backend: Literal['nglview']*) → nglview.NGLWidget

Render a visualization of the molecule in Jupyter

Parameters

- **backend** – The visualization engine to use. Choose from:
  - "rdkit"
  - "openeye"
  - "nglview" (requires conformers)
- **width** – Width of the generated representation (only applicable to backend="openeye" or backend="rdkit")
- **height** – Width of the generated representation (only applicable to backend="openeye" or backend="rdkit")
- **show_all_hydrogens** – 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: str | None = None, strict_chirality: bool = 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** – Path to substructure library file in JSON format. Defaults to using built-in substructure file.
- **strict_chirality** – Whether to use strict chirality symbols (stereomarks) for substructure matchings with SMARTS.

add_default_hierarchy_schemes(*overwrite_existing: bool = 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 – 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.

See also:

`HierarchyScheme`, `Molecule.add_hierarchy_scheme`, `Molecule.update_hierarchy_schemes`, `Molecule.perceive_residues`

add_hierarchy_scheme(*uniqueness_criteria: Iterable[str], iterator_name: str*) → *HierarchyScheme*

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** – 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** – Name of the iterator that will be exposed to access the hierarchy elements generated by this scheme. Must not match an existing attribute of the `Molecule`, i.e. `atoms`, `angles`, etc.

Returns

new_hier_scheme – The newly created HierarchyScheme

See also:

`Molecule.add_default_hierarchy_schemes`, `Molecule.hierarchy_schemes`, `Molecule.delete_hierarchy_scheme`, `Molecule.update_hierarchy_schemes`, `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 – 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: ToolkitRegistry | ToolkitWrapper | None = 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: FrozenMolecule | _SimpleMolecule | Graph, mol2: FrozenMolecule | _SimpleMolecule | Graph, return_atom_map: bool = False, aromatic_matching: bool = True, formal_charge_matching: bool = True, bond_order_matching: bool = True, atom_stereochemistry_matching: bool = True, bond_stereochemistry_matching: bool = True, strip_pyrimidal_n_atom_stereo: bool = True, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY*) → tuple[bool, Optional[dict[int, int]]]

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** – The first molecule to test for isomorphism.
- **mol2** – The second molecule to test for isomorphism.
- **return_atom_map** – Return a `dict` containing the atomic mapping, otherwise `None`. Only processed if inputs are isomorphic, will always return `None` if inputs are not isomorphic.
- **aromatic_matching** – If `False`, aromaticity of graph nodes and edges are ignored for the purpose of determining isomorphism.
- **formal_charge_matching** – If `False`, formal charges of graph nodes are ignored for the purpose of determining isomorphism.
- **bond_order_matching** – If `False`, bond orders of graph edges are ignored for the purpose of determining isomorphism.
- **atom_stereochemistry_matching** – If `False`, atoms' stereochemistry is ignored for the purpose of determining isomorphism.
- **bond_stereochemistry_matching** – If `False`, bonds' stereochemistry is ignored for the purpose of determining isomorphism.
- **strip_pyrimidal_n_atom_stereo** – If `True`, any stereochemistry defined around pyrimidal nitrogen stereocenters will be disregarded in the isomorphism check.
- **toolkit_registry** – `ToolkitRegistry` or `ToolkitWrapper` to use for removing stereochemistry from pyrimidal nitrogens.

Returns

- *molecules_are_isomorphic*
- *atom_map* – [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: str | None = None, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY, use_conformers: Iterable[Quantity] | None = 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** – ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion
- **bond_order_model** – The bond order model to use for fractional bond order calculation. If None, "am1-wiberg" is used.
- **use_conformers** – The conformers to use for fractional bond order calculation. If None, an appropriate number of conformers will be generated by an available ToolkitWrapper.

Examples

```
>>> from openff.toolkit import Molecule
>>> 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: bool = False, use_conformers: Iterable[Quantity] | None = None, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY, normalize_partial_charges: bool = 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"

By default, the conformers on the input molecule are not used in the charge calculation. Instead, any conformers needed for the charge calculation are generated by this method. If this behavior is undesired, specific conformers can be provided via the use_conformers argument.

ELF10 methods will neither fail nor warn when fewer than the expected number of conformers could be generated, as many small molecules are too rigid to provide a large number of conformers. Note that only the "am1bccelf10" partial charge method uses ELF conformer selection; the "am1bcc" method only uses a single conformer. This may confuse users as the ToolkitAM1BCC SMIRNOFF tag in a force field file defines that AM1BCC-ELF10 should be used if the OpenEye Toolkits are available.

For more supported charge methods and their 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** – The partial charge calculation method to use for partial charge calculation.
- **strict_n_conformers** – 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** – Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.
- **toolkit_registry** – ToolkitRegistry or ToolkitWrapper to use for the calculation.
- **normalize_partial_charges** – 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

Generate AM1 Mulliken partial charges. Conformers for the AM1 calculation are generated automatically:

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

To use pre-generated conformations, use the use_conformers argument:

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.generate_conformers(n_conformers=1)
>>> molecule.assign_partial_charges(
...     'am1-mulliken',
...     use_conformers=molecule.conformers
... )
```

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 –

Returns

atom

atom_index(*atom:* Atom) → int

Returns the index of the given atom in this molecule

Parameters
atom –

Returns
index – 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 –

Returns
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.ToolkitWrapper, optional `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

Returns
molecule – An new OpenFF style molecule with atoms in the canonical order.

chemical_environment_matches(*query: str, unique: bool = False, toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY*)

Find matches in the molecule for a SMARTS string

Parameters

- **query** – SMARTS string (with one or more tagged atoms).
- **unique** – If `True`, de-duplicates matches before returning.
- **toolkit_registry** – `ToolkitRegistry` or `ToolkitWrapper` to use for chemical environment matches

Returns
matches – 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]')
```

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: str*)

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 –

See also:

`Molecule.add_hierarchy_scheme`, `Molecule.update_hierarchy_schemes`, `Molecule.hierarchy_schemes`, `HierarchyScheme`

enumerate_protomers(*max_states: int = 0*) → list

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

Parameters

max_states – The maximum number of protomer states to be returned. If 0, the default, attempt to return all protomers. If set to a non-zero number, the input molecule is not guaranteed to be included in the returned list.

Returns

molecules – A list of the protomers of the input molecules, including the input molecule if found by the underlying toolkit's protomer enumeration tool and not pruned by *max_states*.

enumerate_stereoisomers(*undefined_only: bool = False, max_isomers: int = 20, rationalise: bool = True, toolkit_registry:* ToolkitRegistry | ToolkitWrapper *= GLOBAL_TOOLKIT_REGISTRY*)

Enumerate the stereocenters and bonds of the current molecule.

Parameters

- **undefined_only** – If we should enumerate all stereocenters and bonds or only those with undefined stereochemistry
- **max_isomers** – The maximum amount of molecules that should be returned
- **rationalise** – If we should try to build and rationalise the molecule to ensure it can exist
- **toolkit_registry** – `ToolkitRegistry` or `ToolkitWrapper` to use to enumerate the stereoisomers.

Returns

molecules – 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** – The maximum amount of molecules that should be returned
- **toolkit_registry** – `ToolkitRegistry` or `ToolkitWrapper` to use to enumerate the tautomers.

Returns

molecules – A list of openff.toolkit.topology.Molecule instances not including the input molecule.

find_rotatable_bonds(*ignore_functional_groups: list[str] | None = None, toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY*) → list[*Bond*]

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

Parameters

- **ignore_functional_groups** – A list of bond SMARTS patterns to be ignored when finding rotatable bonds.
- **toolkit_registry** – `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 – A BSON serialized representation of the object

Returns

instance – An instantiated object

classmethod from_dict(*molecule_dict: dict*) → FM

Create a new Molecule from a dictionary representation

Parameters

molecule_dict – A dictionary representation of the molecule.

Returns

molecule – A Molecule created from the dictionary representation

classmethod from_file(*file_path: str | Path | TextIO, file_format=None, toolkit_registry=GLOBAL_TOOLKIT_REGISTRY, allow_undefined_stereo: bool = False*) → FM | list[FM]

Create one or more molecules from a file

Parameters

- **file_path** – The path to the file or file-like object to stream one or more molecules from.

- **file_format** – 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** – `ToolkitRegistry` or `ToolkitWrapper` to use for file loading. If a Toolkit is passed, only the highest-precedence toolkit is used
- **allow_undefined_stereo** – If false, raises an exception if oemol contains undefined stereochemistry.

Returns
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 import Molecule
>>> from openff.toolkit.utils.utils import get_data_file_path
>>> sdf_file_path = get_data_file_path("molecules/toluene.sdf")
>>> molecule = Molecule.from_file(sdf_file_path)
```

classmethod from_inchi(*inchi: str, allow_undefined_stereo: bool = False, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY, name: str = ''*) → FM

Construct a Molecule from a InChI representation

Parameters

- **inchi** – The InChI representation of the molecule.
- **allow_undefined_stereo** – 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** – `ToolkitRegistry` or `ToolkitWrapper` to use for InChI-to-molecule conversion
- **name** – An optional name for the output molecule

Returns
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: str, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY, allow_undefined_stereo: bool = False, **kwargs*) → FM

Generate a molecule from IUPAC or common name

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

Parameters

- **iupac_name** – IUPAC name of molecule to be generated
- **toolkit_registry** – `ToolkitRegistry` or `ToolkitWrapper` to use for chemical environment matches
- **allow_undefined_stereo** – If false, raises an exception if molecule contains undefined stereochemistry.

Returns
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 – A JSON serialized representation of the object

Returns
instance – An instantiated object

classmethod from_mapped_smiles(*mapped_smiles: str, toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY, allow_undefined_stereo: bool = False*) → FM

Create a `Molecule` from a SMILES string, ordering atoms from mappings

SMILES strings support mapping integer indices to each atom by ending a bracketed atom declaration with a colon followed by a 1-indexed integer:

This method creates a `Molecule` from such a SMILES string whose atoms are ordered according to the mapping. Each atom must be mapped exactly once; any duplicate, missing, or out-of-range mappings will cause the method to fail.

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

Parameters

- **mapped_smiles** (`str`) – A mapped SMILES string with explicit hydrogens.
- **toolkit_registry** – Cheminformatics toolkit to use for SMILES-to-molecule conversion
- **allow_undefined_stereo** – If false, raise an exception if the SMILES contains undefined stereochemistry.

Returns
offmol – An OpenFF molecule instance.

Raises

- **SmilesParsingError** – If the given SMILES had no indexing picked up by the toolkits, or if the indexing is missing indices.
- **RemapIndexError** – If the mapping has duplicate or out-of-range indices.

Examples

Create a mapped chlorofluoroiodomethane molecule and check the atoms are placed accordingly:

```
>>> molecule = Molecule.from_mapped_smiles(
...     "[Cl:2][C@:1]([F:3])([I:4])[H:5]"
... )
>>> assert molecule.atom(0).symbol == "C"
>>> assert molecule.atom(1).symbol == "Cl"
>>> assert molecule.atom(2).symbol == "F"
>>> assert molecule.atom(3).symbol == "I"
>>> assert molecule.atom(4).symbol == "H"
```

See also:

from_smiles, remap

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

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

Parameters
serialized – A MessagePack-encoded bytes serialized representation

Returns
instance – Instantiated object.

classmethod from_openeye(*oemol*, *allow_undefined_stereo: bool = False*) → *FrozenMolecule*

Create a Molecule from an OpenEye molecule.

Requires the OpenEye toolkit to be installed.

Parameters

- **oemol** – An OpenEye molecule
- **allow_undefined_stereo** – If False, raises an exception if oemol contains undefined stereochemistry.

Returns
molecule – An OpenFF molecule

Examples

Create a `Molecule` from an OpenEye OEMol

```
>>> from openff.toolkit import Molecule
>>> from openeye import oechem
>>> oemol = oechem.OEMol()
>>> oechem.OESmilesToMol(oemol, '[H]C([H])([H])C([H])([H])O[H]')
True
>>> molecule = Molecule.from_openeye(oemol)
```

classmethod from_pdb_and_smiles(*file_path*, *smiles*, *allow_undefined_stereo: bool = False*, *name: str = ''*) → FM

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** – PDB file path
- **smiles** – a valid smiles string for the pdb, used for stereochemistry, formal charges, and bond order
- **allow_undefined_stereo** – If false, raises an exception if SMILES contains undefined stereochemistry.
- **name** – An optional name for the output molecule

Returns
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 – A pickled representation of the object

Returns
instance – An instantiated object

classmethod from_polymer_pdb(*file_path: str | Path | TextIO*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *name: str = ''*) → FM

Loads a polymer from a PDB file.

Also see `Topology.from_multicomponent_pdb()`, which can do everything this method can and more.

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. Only one polymer chain may be present in the PDB file, and it must be the only molecule present.

Connectivity and bond orders are assigned by matching SMARTS codes for the supported residues against atom names. The PDB file must include all atoms with the correct standard atom names described in the PDB Chemical Component Dictionary. Residue names are used to assist troubleshooting failed assignments, but are not used in the actual assignment process.

Metadata such as residues, chains, and atom names are recorded in the `Atom.metadata` attribute, which is a dictionary mapping from strings like "residue_name" 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`.

Parameters

- **file_path** – PDB information to be passed to OpenMM PDBFile object for loading
- **None** (`toolkit_registry = ToolkitWrapper or ToolkitRegistry. Default =`) – Either a ToolkitRegistry, ToolkitWrapper
- **name** – An optional name for the output molecule

Returns

molecule

Raises

- **UnassignedChemistryInPDBError** – If an atom or bond could not be assigned; the exception will provide a detailed diagnostic of what went wrong.
- **MultipleMoleculesInPDBError** – If all atoms and bonds could be assigned, but the PDB includes multiple chains or molecules.

classmethod from_qcschema(*qca_object*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *allow_undefined_stereo: bool = False*)

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

If this method is provided a QCElemental Molecule (or dict representation of a Molecule), it will return a single-conformer OpenFF Molecule.

If this method is provided a QCFractal dataset Entry (or dict representation of an Entry), it will return an OpenFF Molecule with at least one conformer, corresponding to the:

- `.molecule` attribute of a SinglepointDatasetEntry (single conformer)
- `.initial_molecule` attribute of an OptimizationDatasetEntry or GridoptimizationDatasetEntry (single conformer)
- `initial_molecules` attribute of a TorsiondriveDatasetEntry (one or more conformers, in the order that they appear when accessing the `initial_molecules` attribute on the Entry object)

If these QC molecules have their `.id` fields populated, the returned OpenFF Molecule will have a dict mapping QC IDs to conformer numbers (`offmol.properties["initial_molecules"]`)

The data source must also specify the kekule structure of the molecule. Currently the only supported format for this is in the `canonical_isomeric_explicit_hydrogen_mapped_smiles` field, which will be taken from the following locations, if available, in the following order of priority:

- The input's `attributes` attribute (set on QCFractal DatasetEntry objects, such as `SinglepointDatasetEntry` and `TorsiondriveDatasetEntry`)
- The input's `identifiers` attribute (set on QCSchema Molecules made after QCFractal 0.50)
- The input's `extras` attribute (the information was typically set on QCSchema Molecules as part of OpenFF's QC data submission pipeline before QCFractal 0.50)

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

Parameters

- **qca_object** – A QCArchive molecule record or dataset entry, or dict representation of either.
- **toolkit_registry** – openff.toolkit.utils.toolkits.ToolkitWrapper, optional `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion
- **allow_undefined_stereo** – If false, raises an exception if qca_object contains undefined stereochemistry.

Returns

molecule – An OpenFF molecule instance.

Examples

Get Molecule from a QCArchive molecule record:

```
>>> try:
...     from qcportal import PortalClient
... except ImportError:
...     import pytest
...     pytest.skip("This tests sometimes fails when OpenEye is installed")
>>> client = PortalClient("https://api.qcarchive.molssi.org:443/")
>>> offmol = Molecule.from_qcschema(
...     [*client.query_molecules(molecular_formula="C16H20N3O5")][-1]
... )
>>> offmol.to_hill_formula()
'C16H20N3O5'
```

Get Molecule from a QCArchive optimization entry:

```
>>> from qcportal import PortalClient
>>> client = PortalClient("https://api.qcarchive.molssi.org:443/")
>>> optimizations = client.get_dataset(
...     dataset_type="optimization",
...     dataset_name="SMIRNOFF Coverage Set 1",
... )
>>> offmol = Molecule.from_qcschema(optimizations.get_entry('coc(o)oc-0'))
>>> offmol.to_hill_formula()
'C3H8O3'
```

Raises

- **InvalidQCInputError** – If the input record isn't suitable to be made into an OpenFF Molecule
- **MissingCMILESError** – If the record does not contain the `canonical_isomeric_explicit_hydrogen_mapped_smiles`.
- **InvalidConformerError** – If the conformer could not be attached.

classmethod from_rdkit(*rdmol, allow_undefined_stereo: bool = False, hydrogens_are_explicit: bool = False*) → FM

Create a Molecule from an RDKit molecule.

Requires the RDKit to be installed.

Parameters

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

Returns
molecule – An OpenFF molecule

Examples

Create a molecule from an RDKit molecule

```
>>> from openff.toolkit import Molecule
>>> from rdkit import Chem
>>> rdmol = Chem.MolFromSmiles("CCO")
>>> molecule = Molecule.from_rdkit(rdmol)
```

classmethod from_smiles(*smiles: str, hydrogens_are_explicit: bool = False, toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY, allow_undefined_stereo: bool = False, name: str = ''*) → FM

Construct a `Molecule` from a SMILES representation

The order of atoms in the `Molecule` is unspecified and may change from version to version or with different toolkits. SMILES atom indices (also known as atom maps) are not used to order atoms; instead, they are stored in the produced molecule's properties attribute, accessible via `molecule.properties["atom_map"]`. The atom map is stored as a dictionary mapping molecule atom indices to SMILES atom maps. To order atoms according to SMILES atom indices, see `Molecule.from_mapped_smiles()`, which helpfully raises an exception if any atom map is missing, duplicated, or out-of-range, or else `Molecule.remap()` for arbitrary remaps.

Parameters

- **smiles** – The SMILES representation of the molecule.
- **hydrogens_are_explicit** – If `True`, forbid the cheminformatics toolkit from inferring hydrogen atoms not explicitly specified in the SMILES.
- **toolkit_registry** – The cheminformatics toolkit to use to interpret the SMILES.

- **allow_undefined_stereo** – 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.
- **name** – An optional name for the output molecule

Raises

`RadicalsNotSupportedError` – If any atoms in the input molecule contain radical electrons.

Examples

Create a `Molecule` representing toluene from SMILES:

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

Create a `Molecule` representing phenol from SMILES with the oxygen at atom index 0 (SMILES indices begin at 1):

```
>>> molecule = Molecule.from_smiles('c1ccccc1[OH:1]')
>>> molecule = molecule.remap(
...     {k: v - 1 for k, v in molecule.properties["atom_map"].items()},
...     partial=True,
... )
>>> assert molecule.atom(0).symbol == "O"
```

See also:

`from_mapped_smiles`, `remap`

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

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

Parameters

serlialized – A TOML serialized representation of the object

Returns

instance – An instantiated object

classmethod from_topology(*topology*) → FM

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

Parameters

topology – The `Topology` object containing a single `Molecule` object. Note that OpenMM and MDTraj `Topology` objects are not supported.

Returns

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

```
>>> from openff.toolkit import Molecule, Topology
>>> topology = Topology.from_molecules(Molecule.from_smiles('[CH4]'))
>>> 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 – An XML serialized representation

Returns
instance – Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters
serialized – A YAML serialized representation of the object

Returns
instance – Instantiated object

generate_conformers(*toolkit_registry: ToolkitRegistry | ToolkitWrapper = GLOBAL_TOOLKIT_REGISTRY, n_conformers: int = 10, rms_cutoff: Quantity | None = None, clear_existing: bool = True, make_carboxylic_acids_cis: bool = 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** – `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion
- **n_conformers** – The maximum number of conformers to produce
- **rms_cutoff** – 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** – Whether to overwrite existing conformers for the molecule
- **make_carboxylic_acids_cis** – 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 from the element symbol and count.

Names are generated from the elemental symbol and the number of times that element is found in the molecule. 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. For example, generated atom names might begin 'C1x', 'H1x', 'O1x', 'C2x', etc.

get_available_charge_methods(*toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY*) → list[str]

Get the charge methods supported by each wrapper in the specified registry.

Parameters
toolkit_registry – ToolkitRegistry or ToolkitWrapper to use for the calculation.

get_bond_between(*i: int* | Atom, *j: int* | Atom) → *Bond*

Returns the bond between two atoms

Parameters

- **i** – Atoms or atom indices to check
- **j** – Atoms or atom indices to check

Returns
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.

See also:

Molecule.add_hierarchy_scheme, Molecule.delete_hierarchy_scheme, Molecule.update_hierarchy_schemes, Topology.hierarchy_iterator, HierarchyScheme

property hill_formula: str

Get the Hill formula of the molecule

property improperes: set[tuple[Atom, Atom, Atom, Atom]]

Iterate over all improper torsions in the molecule.

Returns

impropers – An iterator of tuples, each containing the atoms making up a possible improper torsion.

See also:

smirnoff_impropers, amber_impropers

is_isomorphic_with(*other:* FrozenMolecule | *_SimpleMolecule* | *Graph*, ***kwargs*) → bool

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** –
- **aromatic_matching** –
- **atoms.** (compare the formal charges attributes of the) –
- **formal_charge_matching** –
- **atoms.** –
- **bond_order_matching** –
- **bonds.** (compare the bond order on attributes of the) –
- **atom_stereochemistry_matching** – If False, atoms' stereochemistry is ignored for the purpose of determining equality.
- **bond_stereochemistry_matching** – If False, bonds' stereochemistry is ignored for the purpose of determining equality.
- **strip_pyrimidal_n_atom_stereo** – If True, any stereochemistry defined around pyrimidal nitrogen stereocenters will be disregarded in the isomorphism check.
- **toolkit_registry** – ToolkitRegistry or ToolkitWrapper to use for removing stereochemistry from pyrimidal nitrogens.

Returns

isomorphic

property n_angles: int

Number of angles in the molecule.

property n_atoms: int

The number of Atom objects.

property n_bonds: int

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_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 – 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() → int

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 – The partial charges on the molecule's atoms. Returns None if no charges have been specified.

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: dict[int, int], current_to_new: bool = True, partial: bool = False*)

Reorder the atoms in the molecule according to the given mapping dict.

The mapping dict must be a dictionary mapping atom indices to atom indices. Each atom index must be an integer in the half-open interval `[0, n_atoms)`; ie, it must be a valid index into the `self.atoms` list. All atom indices in the molecule must be mapped from and to exactly once unless `partial=True` is given, in which case they must be mapped no more than once. Missing (unless `partial=True`), out-of-range (including non-integer), or duplicate indices are not allowed in the `mapping_dict` and will lead to an exception.

By default, the mapping dict's keys are the source indices and its values are destination indices, but this can be changed with the `current_to_new` argument.

The keys of the `self.properties["atom_map"]` property are updated for the new ordering. Other values of the properties dictionary are transferred unchanged.

Warning: This API is experimental and subject to change.

Parameters

- **mapping_dict** – A dictionary of the mapping between indices. The mapping should be indexed starting from 0 for both the source and destination; note that SMILES atom mapping is typically 1-based.
- **current_to_new** – 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}`.
- **partial** – If `False` (the default), an exception will be raised if any atom is lacking a destination in the atom map. Note that if this is `True`, atoms without entries in the mapping dict may be moved in addition to those in the dictionary. Note that partial maps must still be in-range and not include duplicates.

Returns

new_molecule – A copy of the molecule in the new order.

Raises

RemapIndexError – When an out-of-range, duplicate, or missing index is found in the `mapping_dict`.

See also:

`from_mapped_smiles`

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 – 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: str*, *toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *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) – Tagged SMARTS with a single atom with index 1. Any matches for this atom will have any assigned stereocheistry information removed.
- **toolkit_registry** – ToolkitRegistry or ToolkitWrapper to use for I/O operations

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized – A BSON serialized representation of the objecft

to_dict() → dict

Return a dictionary representation of the molecule.

Returns

molecule_dict – 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** – A file-like object or the path to the file to be written.
- **file_format** – Format specifier, one of ['MOL2', 'MOL2H', 'SDF', 'PDB', 'SMI', 'CAN', 'TDT'] Note that not all toolkits support all formats
- **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.

to_inchi(*fixed_hydrogens: bool = False*, *toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY*) → str

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** – 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** – `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: bool = False, toolkit_registry:* ToolkitRegistry | ToolkitWrapper = *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** – 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** – `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* – 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 – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – A JSON serialized representation of the object

to_messagepack()

Return a MessagePack representation.

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

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

to_networkx() → Graph

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 – 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:* ToolkitRegistry | ToolkitWrapper = *GLOBAL_TOOLKIT_REGISTRY*, *aromaticity_model:* *str* = *DEFAULT_AROMATICITY_MODEL*)

Create an OpenEye molecule

Requires the OpenEye toolkit to be installed.

Parameters
aromaticity_model – The aromaticity model to use. Only OEAroModel_MDL is supported.

Returns
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 – A pickled representation of the object

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

Create a QCElemental Molecule.

The kekule structure of the molecule is saved in two places on the returned Molecule:

- `extras["canonical_isomeric_explicit_hydrogen_mapped_smiles"]`
- `identifiers["canonical_isomeric_explicit_hydrogen_mapped_smiles"]`

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

Parameters

- **multiplicity** – The multiplicity of the molecule; sets `molecular_multiplicity` field for QCElemental Molecule.
- **conformer** – The index of the conformer to use for the QCElemental Molecule geometry.
- **extras** – 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*) → RDMol

Create an RDKit molecule

Requires the RDKit to be installed.

Parameters

aromaticity_model – The aromaticity model to use. Only OEAroModel_MDL is supported.

Returns

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: bool = True, explicit_hydrogens: bool = True, mapped: bool = False, toolkit_registry: ToolkitRegistry | ToolkitWrapper = 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** – return an isomeric smiles
- **explicit_hydrogens** – return a smiles string containing all hydrogens explicitly
- **mapped** – 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** – `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES conversion

Returns

smiles – 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 – A TOML serialized representation of the object

to_topology()

Return an OpenFF Topology representation containing one copy of this molecule

Returns
topology – A Topology representation of this molecule

Examples

```
>>> from openff.toolkit import Molecule
>>> 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 – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – A MessagePack-encoded bytes serialized representation.

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns
serialized – 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

property total_charge

Return the total charge on the molecule

update_hierarchy_schemes(*iter_names: list[str] | None = 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 – Only perceive hierarchy for HierarchySchemes that expose these iterator names. If not provided, all known hierarchies will be perceived, overwriting previous results if applicable.

See also:

`Molecule.add_hierarchy_scheme`, `Molecule.delete_hierarchy_schemes`, `Molecule.hierarchy_schemes`, `HierarchyScheme`

12.1.3 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 – 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)	Add a copy of the molecule to the topology
`assert_bonded`(atom1, atom2)	Raise an exception if the specified atoms are not bonded in the topology.
`atom`(atom_topology_index)	Get the Atom 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 Bond at a given Topology bond index.
`chemical_environment_matches`(query[, ...])	Retrieve all matches for a given chemical environment query.
`clear_positions`()	Clear the positions of this topology by removing all conformers from its consituent molecules.
`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[, ...])	Construct an OpenFF `Topology` from an MDTraj `Topology`
`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[, ...])	Construct an OpenFF Topology object from an OpenMM Topology object.
`from_pdb`(file_path[, unique_molecules, ...])	Loads supported or user-specified molecules from a PDB file.
`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
`get_positions`()	Copy the positions of the topology into a new array.
`hierarchy_iterator`(iter_name)	Iterate over all molecules with the given hierarchy scheme.
`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.

continues on next page

Table 3 – continued from previous page

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.
set_positions(array)	Set the positions in a topology by copying from a single n×3 array.
to_bson()	Return a BSON serialized representation.
to_dict()	
to_file(file[, 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.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.
visualize([ensure_correct_connectivity])	Visualize with NGLView.

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	The number of possible improper torsions in this Topology.
n_molecules	Returns the number of molecules in this Topology
n_propers	The number of proper torsions in this Topology.
n_unique_molecules	Returns the number of unique molecules in this Topology
propers	iterator over the proper torsions in this Topology.
smirnoff_impropers	Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom second in each improper.
unique_molecules	Get a list of chemically unique molecules in this Topology.

property unique_molecules: Iterator[Molecule | _SimpleMolecule]

Get a list of chemically unique molecules in this Topology.

Molecules are considered unique if they fail an isomorphism check with default values (see `Molecule.is_isomorphic_with()`). The order of molecules returned by this property is arbitrary.

property n_unique_molecules: int

Returns the number of unique molecules in this Topology

classmethod from_molecules(*molecules:* Molecule | FrozenMolecule | *_SimpleMolecule* | *list[Union[*Molecule, FrozenMolecule, *openff.toolkit.topology._mm_molecule._SimpleMolecule]]*) → *Topology*

Create a new Topology object containing one copy of each of the specified molecule(s).

Parameters
molecules – One or more molecules to be added to the Topology

Returns
topology – The Topology created from the specified molecule(s)

assert_bonded(*atom1: int* | Atom, *atom2: int* | Atom)

Raise an exception if the specified atoms are not bonded in the topology.

Parameters

- **atom1** – The atoms or atom topology indices to check to ensure they are bonded
- **atom2** – The atoms or atom topology indices to check to ensure they are bonded

property aromaticity_model: str

Get the aromaticity model applied to all molecules in the topology.

Returns
aromaticity_model – Aromaticity model in use.

property box_vectors

Return the box vectors of the topology, if specified

Returns
box_vectors – The unit-wrapped box vectors of this topology

property is_periodic: bool

Return whether or not this Topology is intended to be described with periodic boundary conditions.

property constrained_atom_pairs: dict[tuple[int, int], Union[Quantity, bool]]

Returns the constrained atom pairs of the Topology

Returns
constrained_atom_pairs – dictionary of the form {(atom1_topology_index, atom2_topology_index): distance}

property n_molecules: int

Returns the number of molecules in this Topology

property molecules: Generator[Molecule | FrozenMolecule | _SimpleMolecule, None, None]

Returns an iterator over all the Molecules in this Topology

Returns
molecules

molecule(*index: int*) → *Molecule* | _SimpleMolecule

Returns the molecule with a given index in this Topology.

Returns
molecule

property n_atoms: int

Returns the number of atoms in in this Topology.

Returns
n_atoms

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

atom_index(*atom:* Atom) → int

Returns the index of a given atom in this topology

Parameters
atom –

Returns
index – The index of the given atom in this topology

Raises
AtomNotInTopologyError – If the given atom is not in this topology

molecule_index(*molecule:* Molecule | FrozenMolecule | *_SimpleMolecule*) → int

Returns the index of a given molecule in this topology

Parameters
molecule –

Returns
index – 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:* Molecule) → int

Returns the index of a molecule's first atom in this topology

Parameters
molecule –

Returns
index

property n_bonds: int

Returns the number of Bonds in in this Topology.

Returns
n_bonds

property bonds: Generator[Bond, None, None]

Returns an iterator over the bonds in this Topology

Returns
bonds

property n_angles: int

number of angles in this Topology.

Type
int

property angles: Generator[tuple['Atom', ...], None, None]

Iterator over the angles in this Topology. Returns a Generator of tuple[Atom].

property n_propers: int

The number of proper torsions in this Topology.

property propers:], None, None]

iterator over the proper torsions in this Topology.

Type
Iterable of *tuple*[Atom]

property n_impropers: int

The number of possible improper torsions in this Topology.

property impropers: Generator[tuple['Atom', ...], None, None]

iterator over the possible improper torsions in this Topology.

Type
Generator of *tuple*[Atom]

property smirnoff_impropers:], None, None]

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
smirnoff_impropers – An iterator of tuples, each containing the Atom objects comprising up a possible improper torsion. The central atom is listed second in each tuple.

See also:

`impropers`, `amber_impropers`

property amber_impropers:], None, None]

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
amber_impropers – An iterator of tuples, each containing the Atom objects comprising 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: str, aromaticity_model: str = 'MDL', unique: bool = False, toolkit_registry:* ToolkitRegistry | ToolkitWrapper *= 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** – SMARTS string (with one or more tagged atoms)
- **aromaticity_model** – Override the default aromaticity model for this topology and use the specified aromaticity model instead. Allowed values: ['MDL']

Returns
matches – A list of tuples, containing the topology indices of the matching atoms.

property identical_molecule_groups: dict[int, list[tuple[int, dict[int, int]]]]

Returns groups of chemically identical molecules, identified by index and atom map.

Returns
identical_molecule_groups – A dict of the form `{unique_mol_idx : [(topology_mol_idx, atom_map), ...]`. Each key is the topology molecule index of the first instance of a unique chemical species. Iterating over the keys will yield all of the unique chemical species in the topology. Each value is a list of all instances of that chemical species in the topology. Each element of the list is a 2-tuple where the first element is the molecule index of the instance, and the second element maps the atom topology indices of the key molecule to the instance. The molecule instance corresponding to the key is included in the list, so the list is a complete list of all instances of that chemical species.

Examples

```
>>> from openff.toolkit import Molecule, Topology
>>> # Create a water ordered as OHH
>>> water1 = Molecule()
>>> water1.add_atom(8, 0, False)
0
>>> water1.add_atom(1, 0, False)
1
>>> water1.add_atom(1, 0, False)
2
>>> _ = 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)
0
>>> water2.add_atom(8, 0, False)
1
>>> water2.add_atom(1, 0, False)
2
>>> _ = 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: dict*)

Create a new Topology from a dictionary representation

Parameters

topology_dict – A dictionary representation of the topology.

Returns

topology – A Topology created from the dictionary representation

classmethod from_openmm(*openmm_topology: openmm.app.Topology, unique_molecules: Iterable[FrozenMolecule] | None = None, positions: None | Quantity | OMMQuantity = None*) → *Topology*

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.

Hierarchy schemes are taken from the OpenMM topology, not from `unique_molecules`.

If any virtual sites are detected in the OpenMM topology, `VirtualSitesUnsupportedError` is raised because the `Topology` object model does not store virtual sites.

Parameters

- **openmm_topology** – The OpenMM Topology object to convert
- **unique_molecules** – An iterable containing all the unique molecules in the topology. This is used to identify the molecules in the OpenMM topology and provide

any missing chemical information. Each chemical species in the topology must be specified exactly once, though the topology may have any number of copies, including zero. The chemical elements of atoms and their bond connectivity will be used to match these reference molecules to the molecules appearing in the topology. If bond orders are specified in the topology, these will be used in matching as well.

- **positions** – Positions for the atoms in the new topology.

Returns

topology – An OpenFF Topology object, constructed from the molecules in `unique_molecules`, with the same atom order as the input topology.

Raises

- **MissingUniqueMoleculesError** – If `unique_molecules` is `None`
- **DuplicateUniqueMoleculeError** – If the same connectivity graph is represented by two different molecules in `unique_molecules`
- **ValueError** – If a chemically impossible molecule is detected in the topology

classmethod from_pdb(*file_path: str | Path | TextIO, unique_molecules: Iterable[Molecule] | None = None, toolkit_registry=GLOBAL_TOOLKIT_REGISTRY, _custom_substructures: dict[str, list[str]] | None = None, _additional_substructures: Iterable[Molecule] | None = None*)

Loads supported or user-specified molecules from a PDB file.

Currently, canonical proteins, waters, and monoatomic ions are supported without CONECT records via residue and atom names, and molecules specified in the `unique_molecules` argument are supported when CONECT records are provided.

> **Warning:** Molecules in the resulting Topology will adopt the geometric stereochemistry in the PDB, even if this conflicts with the stereochemistry specified in `unique_molecules`.

All molecules in the PDB file have the following requirements:

- Polymer molecules must use the standard atom names described in the PDB Chemical Component Dictionary (PDB CCD).
- There must be no missing atoms (all hydrogens must be explicit).
- All particles must correspond to an atomic nucleus (particles in the PDB representing "virtual sites" or "extra points" are not allowed).
- All bonds must be specified by either CONECT records, or for polymers and water by the PDB CCD via the residue and atom name.
- CONECT records must correspond only to chemical bonds (CONECT records representing an angle constraints are not allowed).
- CONECT records may be redundant with connectivity defined by residue templates.

Currently, the only supported polymers are proteins made of the 20 canonical amino acids. For details on the polymer loading used here, see `Molecule.from_polymer_pdb()`.

Waters can be recognized in either of two ways:

- By the residue name "HOH" and atom names "H1", "H2", and "O".
- By ATOM records which include element information and CONECT records.

Monoatomic ions supported by Sage are recognized (Na+, Li+, K+, Rb+, Cs+, F-, Cl-, Br-, and I-). To load other monoatomic ions, use the `unique_molecules` keyword argument.

The `unique_molecules` keyword argument can be used to load arbitrary molecules from the PDB file. These molecules match a group of atoms in the PDB file when their atomic elements and connectivity are identical; elements and CONECT records must therefore be explicitly specified in the PDB file. Information missing from the PDB format, such as bond orders and formal charges, is then taken from the matching unique molecule. Unique molecule matches will overwrite bond order and formal charge assignments from other sources. Stereochemistry is assigned based on the geometry in the PDB, even if this differs from the stereochemistry in the unique molecule.

A user-defined molecule in the PDB file must exactly match a unique molecule to successfully load it - substructures and superstructures will raise `UnassignedChemistryInPDBError`. Unique molecules need not be present in the PDB.

Metadata such as residues, chains, and atom names are recorded in the `Atom.metadata` attribute, which is a dictionary mapping from the strings `"residue_name"`, `"residue_number"`, `"insertion_code"`, and `"chain_id"` to the appropriate value. The topology returned by this method can expose residue and chain iterators which can be accessed using `Topology.hierarchy_iterator()`, such as `top.hierarchy_iterator("residues")` and `top.hierarchy_iterator ("chains")`.

A CRYST line in the PDB, if present, will be interpreted as periodic box vectors in Angstroms.

Parameters

- **file_path** – PDB information to be passed to OpenMM PDBFile object for loading
- **unique_molecules** – OpenFF Molecule objects corresponding to the molecules in the input PDB. See above for details.
- **toolkit_registry** – The ToolkitRegistry to use as the cheminformatics backend.
- **_custom_substructures** (`dict[str, list[str]], Default = {}`) – Experimental and unstable. Dictionary where keys are the names of new substructures (cannot overlap with existing amino acid names) and the values are the new substructure entries that follow the same format as those used in the amino acid substructure library
- **_additional_substructures** – Experimental and unstable. Molecule with atom.metadata["substructure_atom"] = True or False for all atoms. Currently only stable for independent, standalone molecules not bonded to a larger protein/molecule. (For that use _custom_substructures)

Returns

topology

Raises

`UnassignedChemistryInPDBError` – If an atom or bond could not be assigned; the exception will provide a detailed diagnostic of what went wrong.

Examples

```
>>> from openff.toolkit import Topology
>>> from openff.toolkit.utils import get_data_file_path
>>> top = Topology.from_pdb(get_data_file_path("proteins/TwoMol_SER_CYS.pdb"))
>>> # The molecules in the loaded topology are full-fledged OpenFF Molecule objects
>>> for match in top.chemical_environment_matches('[O:1]=[C:2][N:3][H:4]'):␣
↪print(match.topology_atom_indices)
(1, 0, 6, 13)
(9, 8, 17, 19)
(24, 23, 29, 36)
(32, 31, 40, 42)
>>> [*top.hierarchy_iterator("residues")]
[HierarchyElement ('A', '1', ' ', 'ACE') of iterator 'residues' containing 6␣
↪atom(s),
 HierarchyElement ('A', '2', ' ', 'SER') of iterator 'residues' containing 11␣
↪atom(s),
 HierarchyElement ('A', '3', ' ', 'NME') of iterator 'residues' containing 6␣
↪atom(s),
 HierarchyElement ('B', '1', ' ', 'ACE') of iterator 'residues' containing 6␣
↪atom(s),
 HierarchyElement ('B', '2', ' ', 'CYS') of iterator 'residues' containing 11␣
↪atom(s),
 HierarchyElement ('B', '3', ' ', 'NME') of iterator 'residues' containing 6␣
↪atom(s)]
```

Polymer systems can also be supported if `_custom_substructures` are given as a `dict[str, list[str]]`, where the keys are unique atom names and the values are lists of substructure SMARTS. The substructure SMARTS must follow the same format as given in the residue substructure connectivity library: `"<bond>[#<atomic number>D<degree>+<formal charge>:<id>]<bond>"` for monomer atoms and `"<bond>[*:<id>]"` for adjacent neighboring atoms (NOTE: This functionality is experimental!)

```
>>> PE_substructs = {
...     "PE": [
...         "[#6D4+0:2](-[#1D1+0:3])(-[#1D1+0:4])(-[#6D4+0:5](-[#1D1+0:6])(-[
↪#1D1+0:7])-[*:8])-[*:1]",
...         "[#6D4+0:2](-[#1D1+0:3])(-[#1D1+0:4])(-[#6D4+0:5](-[#1D1+0:6])(-[
↪#1D1+0:7])-[#1D1+0:8])-[*:1]",
...         "[#6D4+0:2](-[#1D1+0:3])(-[#1D1+0:4])(-[#6D4+0:5](-[#1D1+0:6])(-[
↪#1D1+0:7])-[*:8])-[#1D1+0:1]",
...     ]
... }
>>> top = Topology.from_pdb(
...         get_data_file_path("systems/test_systems/PE.pdb"),
...         _custom_substructures=PE_substructs,
...     )
```

to_openmm(*ensure_unique_atom_names: str | bool = 'residues'*) → openmm.app.Topology

Create an OpenMM Topology object.

The atom metadata fields *residue_name*, *residue_number*, *insertion_code*, and *chain_id* are used to group atoms into OpenMM residues and chains.

Contiguously-indexed atoms with the same *residue_name*, *residue_number*, *insertion_code*, 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 – Whether to generate new atom names to ensure uniqueness within a molecule or hierarchy element.

- If the name of a `HierarchyScheme` is given as a string, new atom names will be generated so that each element of that scheme has unique atom names. Molecules without the given hierarchy scheme will be given unique atom names within that molecule.
- If `True`, new atom names will be generated so that atom names are unique within a molecule.
- If `False`, the existing atom names will be used.

Returns

openmm_topology – An OpenMM Topology object

to_file(*file: Path | str | TextIO, positions: OMMQuantity | Quantity | ndarray[Any, dtype[_ScalarType_co]] | None = None, file_format: Literal['PDB'] = 'PDB', keep_ids: bool = False, ensure_unique_atom_names: str | bool = 'residues'*)

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

- **file** – A file-like object to write to, or a path to save the file to.
- **positions** – May 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 dimensions of length
  - (unitless) 2D `numpy.ndarray`, in which it is assumed that the positions are in units of Angstroms.
  - `None` (the default), in which case the first conformer of each molecule in the topology will be used.
- **file_format** – Output file format. Case insensitive. Currently only supported values are `"PDB"`.

- **keep_ids** – If `True`, keep the residue and chain IDs specified in the Topology rather than generating new ones.
- **ensure_unique_atom_names** – Whether to generate new atom names to ensure uniqueness within a molecule or hierarchy element.
  - If the name of a `HierarchyScheme` is given as a string, new atom names will be generated so that each element of that scheme has unique atom names. Molecules without the given hierarchy scheme will be given unique atom names within that molecule.
  - If `True`, new atom names will be generated so that atom names are unique within a molecule.
  - If `False`, the existing atom names will be used.

  Note that this option cannot guarantee name uniqueness for formats like PDB that truncate long atom names.

get_positions() → Quantity | None

Copy the positions of the topology into a new array.

Topology positions are stored as the first conformer of each molecule. If any molecule has no conformers, this method returns `None`. Note that modifying the returned array will not update the positions in the topology. To change the positions, use `Topology.set_positions()`.

See also:

`set_positions`, `clear_positions`

clear_positions()

Clear the positions of this topology by removing all conformers from its consituent molecules.

Note that all conformers will be deleted (in-place) from all molecules. Use *Topology.get_positions()* if you wish to save them before clearing.

See also:

`get_positions`, `set_positions`

set_positions(*array: Quantity*)

Set the positions in a topology by copying from a single n×3 array.

Note that modifying the original array will not update the positions in the topology; it must be passed again to `set_positions()`.

Parameters

array – Positions for the topology. Should be a unit-wrapped array-like object with shape (n_atoms, 3) and dimensions of length.

See also:

`get_positions`, `clear_positions`

classmethod from_mdtraj(*mdtraj_topology: Topology, unique_molecules: Iterable[*Molecule | FrozenMolecule | *_SimpleMolecule] | None = None, positions: None | OMMQuantity | Quantity = None*)

Construct an OpenFF `Topology` from an MDTraj `Topology`

This method guarantees that the order of atoms in the input MDTraj 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.

Hierarchy schemes are taken from the MDTraj topology, not from `unique_molecules`.

Parameters

- **mdtraj_topology** – The MDTraj Topology object to convert
- **unique_molecules** – An iterable containing all the unique molecules in the topology. This is used to identify the molecules in the MDTraj topology and provide any missing chemical information. Each chemical species in the topology must be specified exactly once, though the topology may have any number of copies, including zero. The chemical elements of atoms and their bond connectivity will be used to match these reference molecules to the molecules appearing in the topology. If bond orders are specified in the topology, these will be used in matching as well.
- **positions** – Positions for the atoms in the new topology.

Returns

topology – An OpenFF Topology object, constructed from the molecules in `unique_molecules`, with the same atom order as the input topology.

Raises

- **MissingUniqueMoleculesError** – If `unique_molecules` is `None`
- **DuplicateUniqueMoleculeError** – If the same connectivity graph is represented by two different molecules in `unique_molecules`
- **ValueError** – If a chemically impossible molecule is detected in the topology

get_bond_between(*i: int* | Atom, *j: int* | Atom) → *Bond*

Returns the bond between two atoms

Parameters

- **i** – Atoms or atom indices to check
- **j** – Atoms or atom indices to check

Returns

bond – The bond between i and j.

is_bonded(*i: int* | Atom, *j: int* | Atom) → bool

Returns True if the two atoms are bonded

Parameters

- **i** – Atoms or atom indices to check
- **j** – Atoms or atom indices to check

Returns

is_bonded – True if atoms are bonded, False otherwise.

atom(*atom_topology_index: int*) → *Atom*

Get the Atom at a given Topology atom index.

Parameters

atom_topology_index – The index of the Atom in this Topology

Returns

An openff.toolkit.topology.Atom

bond(*bond_topology_index: int*) → *Bond*

Get the Bond at a given Topology bond index.

Parameters
bond_topology_index – The index of the Bond in this Topology

Returns
An openff.toolkit.topology.Bond

add_molecule(*molecule:* Molecule | FrozenMolecule | *_SimpleMolecule*) → int

Add a copy of the molecule to the topology

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** – Atoms to mark as constrained These atoms may be bonded or not in the Topology
- **jatom** – Atoms to mark as constrained These atoms may be bonded or not in the Topology
- **distance** – 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** – Indices of atoms to mark as constrained.
- **jatom** – Indices of atoms to mark as constrained.

Returns
distance – True if constrained but constraints have not yet been applied Distance if constraint has already been added to Topology

classmethod from_bson(*serialized*)

Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters
serialized – A BSON serialized representation of the object

Returns
instance – An instantiated object

classmethod from_json(*serialized: str*)

Instantiate an object from a JSON serialized representation.

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

Parameters
serialized – A JSON serialized representation of the object

Returns
instance – An instantiated object

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

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

Parameters
serialized – A MessagePack-encoded bytes serialized representation

Returns
instance – 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 – A pickled representation of the object

Returns
instance – An instantiated object

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

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

Parameters
serlialized – A TOML serialized representation of the object

Returns
instance – An instantiated object

classmethod from_xml(*serialized*)

Instantiate an object from an XML serialized representation.

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

Parameters
serialized – An XML serialized representation

Returns
instance – Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters
serialized – A YAML serialized representation of the object

Returns
instance – Instantiated object

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns
serialized – A BSON serialized representation of the objecft

to_json(*indent=None*) → str

Return a JSON serialized representation.

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

Parameters
indent – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – A JSON serialized representation of the object

to_messagepack()

Return a MessagePack representation.

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

Returns
serialized – 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 – A pickled representation of the object

to_toml()

Return a TOML serialized representation.

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

Returns
serialized – A TOML serialized representation of the object

to_xml(*indent=2*)

Return an XML representation.

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

Parameters
indent – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – A MessagePack-encoded bytes serialized representation.

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns
serialized – A YAML serialized representation of the object

visualize(*ensure_correct_connectivity: bool = False*) → NGLWidget

Visualize with NGLView.

Requires all molecules in this topology have positions.

NGLView is a 3D molecular visualization library for use in Jupyter notebooks. Note that for performance reasons, by default the visualized connectivity is inferred from positions and may not reflect the connectivity in the `Topology`.

Parameters

ensure_correct_connectivity – If `True`, the visualization will be guaranteed to reflect the connectivity in the `Topology`. Note that this will severely degrade performance, especially for topologies with many atoms.

Examples

Visualize a complex PDB file

```
>>> from openff.toolkit import Topology
>>> from openff.toolkit.utils.utils import get_data_file_path
>>> pdb_filename = get_data_file_path("systems/test_systems/T4_lysozyme_water_ions.
↪pdb")
>>> topology = Topology.from_pdb(pdb_filename)
>>> topology.visualize()
```

hierarchy_iterator(*iter_name: str*) → Iterator[*HierarchyElement*]

Iterate over all molecules with the given hierarchy scheme.

Get an iterator over hierarchy elements from all of the molecules in this topology that provide the appropriately named iterator. This iterator will yield hierarchy elements sorted first by the order that molecules are listed in the Topology, and second by the specific sorting of hierarchy elements defined in each molecule. Molecules without the named iterator are not included.

Parameters

iter_name – The iterator name associated with the HierarchyScheme to retrieve (for example 'residues' or 'chains')

Returns

iterator of `HierarchyElement`

See also:

`HierarchyScheme`, `HierarchyElement`, `Molecule.hierarchy_schemes`

12.2 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.

12.2.1 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: FrozenMolecule

The Molecule this particle is part of.

property molecule_particle_index: int

Returns the index of this particle in its molecule

property name: str

The name of the particle

to_dict() → dict

Convert to dictionary representation.

classmethod from_dict(*d: dict*) → P

Static constructor from dictionary representation.

classmethod from_bson(*serialized*)

Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters
serialized – A BSON serialized representation of the object

Returns
instance – An instantiated object

classmethod from_json(*serialized: str*)

Instantiate an object from a JSON serialized representation.

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

Parameters
serialized – A JSON serialized representation of the object

Returns
instance – An instantiated object

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

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

Parameters
serialized – A MessagePack-encoded bytes serialized representation

Returns
instance – 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 – A pickled representation of the object

Returns
instance – An instantiated object

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

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

Parameters
serlialized – A TOML serialized representation of the object

Returns
instance – An instantiated object

classmethod from_xml(*serialized*)

Instantiate an object from an XML serialized representation.

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

Parameters
serialized – An XML serialized representation

Returns
instance – Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters
serialized – A YAML serialized representation of the object

Returns
instance – Instantiated object

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns
serialized – A BSON serialized representation of the objecft

to_json(*indent=None*) → str

Return a JSON serialized representation.

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

Parameters
indent – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – A JSON serialized representation of the object

to_messagepack()

Return a MessagePack representation.

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

Returns
serialized – 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 – A pickled representation of the object

to_toml()

Return a TOML serialized representation.

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

Returns
serialized – A TOML serialized representation of the object

to_xml(*indent=2*)

Return an XML representation.

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

Parameters
indent – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – A MessagePack-encoded bytes serialized representation.

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns
serialized – A YAML serialized representation of the object

12.2.2 Atom

class openff.toolkit.topology.**Atom**(*atomic_number: int, formal_charge: int | Quantity, is_aromatic: bool, name: str | None = None, molecule=None, stereochemistry: str | None = None, metadata: dict[str, Union[int, str]] | None = None*)

A chemical atom.

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

__init__(*atomic_number: int, formal_charge: int | Quantity, is_aromatic: bool, name: str | None = None, molecule=None, stereochemistry: str | None = None, metadata: dict[str, Union[int, str]] | None = None*)

Create an immutable Atom object.

Object is serializable and immutable.

Parameters

- **atomic_number** – Atomic number of the atom. Must be non-negative and non-zero.
- **formal_charge** – Formal charge of the atom
- **is_aromatic** – If True, atom is aromatic; if False, not aromatic
- **stereochemistry** – Either 'R' or 'S' for specified stereochemistry, or None for ambiguous stereochemistry
- **name** – An optional name to be associated with the atom
- **metadata** – 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:* Bond)

Adds a bond that this atom is involved in

Parameters

bond – A bond involving this atom

to_dict() → dict[str, Union[NoneType, str, int, bool, dict[Any, Any]]]

Return a dict representation of the atom.

classmethod from_dict(*atom_dict: dict*) → A

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: Generator[Atom, None, None]

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 – a different atom in the same molecule

Returns
bool – Whether this atom is bound to atom2

is_in_ring(*toolkit_registry: ToolkitRegistry = 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` to use to enumerate the tautomers.

property molecule_atom_index: int

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 – A BSON serialized representation of the object

Returns
instance – An instantiated object

classmethod from_json(*serialized: str*)

Instantiate an object from a JSON serialized representation.

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

Parameters
serialized – A JSON serialized representation of the object

Returns
instance – An instantiated object

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

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

Parameters
serialized – A MessagePack-encoded bytes serialized representation

Returns
instance – 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 – A pickled representation of the object

Returns
instance – An instantiated object

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

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

Parameters
serlialized – A TOML serialized representation of the object

Returns
instance – An instantiated object

classmethod from_xml(*serialized*)

Instantiate an object from an XML serialized representation.

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

Parameters
serialized – An XML serialized representation

Returns
instance – Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters
serialized – A YAML serialized representation of the object

Returns
instance – Instantiated object

property molecule: FrozenMolecule

The `Molecule` this particle is part of.

property molecule_particle_index: int

Returns the index of this particle in its molecule

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns
serialized – A BSON serialized representation of the objecft

to_json(*indent=None*) → str

Return a JSON serialized representation.

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

Parameters
indent – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – A JSON serialized representation of the object

to_messagepack()

Return a MessagePack representation.

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

Returns
serialized – 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 – A pickled representation of the object

to_toml()

Return a TOML serialized representation.

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

Returns
serialized – A TOML serialized representation of the object

to_xml(*indent=2*)

Return an XML representation.

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

Parameters
indent – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – A MessagePack-encoded bytes serialized representation.

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized – A YAML serialized representation of the object

12.2.3 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** – Atoms involved in the bond
- **bond_order** – The (integer) bond order of this bond.
- **is_aromatic** – Whether or not this bond is aromatic.
- **fractional_bond_order** – The fractional bond order, or partial bond order of this bond.
- **stereochemstry** – 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() → dict[str, Union[int, bool, str, float]]

Return a dict representation of the bond.

classmethod from_dict(*molecule: FM*, *d: dict*) → B

Create a Bond from a dict representation.

property molecule_bond_index: int

The index of this Bond within the the list of bonds in `Molecules`.

is_in_ring(*toolkit_registry:* ToolkitRegistry = *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` to use to enumerate the tautomers.

Returns
is_in_ring – 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 – A BSON serialized representation of the object

Returns
instance – An instantiated object

classmethod from_json(*serialized: str*)

Instantiate an object from a JSON serialized representation.

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

Parameters
serialized – A JSON serialized representation of the object

Returns
instance – An instantiated object

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

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

Parameters
serialized – A MessagePack-encoded bytes serialized representation

Returns
instance – 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 – A pickled representation of the object

Returns
instance – An instantiated object

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

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

Parameters
serlialized – A TOML serialized representation of the object

Returns
instance – An instantiated object

classmethod from_xml(*serialized*)

Instantiate an object from an XML serialized representation.

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

Parameters
serialized – An XML serialized representation

Returns
instance – Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters
serialized – A YAML serialized representation of the object

Returns
instance – Instantiated object

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns
serialized – A BSON serialized representation of the objecft

to_json(*indent=None*) → str

Return a JSON serialized representation.

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

Parameters
indent – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – A JSON serialized representation of the object

to_messagepack()

Return a MessagePack representation.

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

Returns
serialized – 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 – A pickled representation of the object

to_toml()

Return a TOML serialized representation.

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

Returns
serialized – A TOML serialized representation of the object

to_xml(*indent=2*)

Return an XML representation.

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

Parameters
indent – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – A MessagePack-encoded bytes serialized representation.

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns
serialized – A YAML serialized representation of the object

12.2.4 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.

classmethod index_of(*key: Iterable[int], possible: Iterable[Iterable[int]] | None = None*) → int

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** – A valid key for ValenceDict
- **possible** – A subset of the possible orderings that this match might take.

Returns

index

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

12.2.5 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.

classmethod 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** – A valid key for ValenceDict
- **possible** – A subset of the possible orderings that this match might take.

Returns

index

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

12.2.6 HierarchyScheme

class openff.toolkit.topology.**HierarchyScheme**(*parent:* Molecule | FrozenMolecule | *_SimpleMolecule*, *uniqueness_criteria: Iterable[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.

See also:

`Molecule.add_default_hierarchy_schemes`, `Molecule.add_hierarchy_scheme`, `Molecule.hierarchy_schemes`, `Molecule.delete_hierarchy_scheme`, `Molecule.update_hierarchy_schemes`, `Molecule.perceive_residues`, `Topology.hierarchy_iterator`, `HierarchyElement`

__init__(*parent:* Molecule | FrozenMolecule | *_SimpleMolecule*, *uniqueness_criteria: Iterable[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() → 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: tuple[Union[str, int]], atom_indices: Sequence[int]*) → *HierarchyElement*

Instantiate a new HierarchyElement belonging to this HierarchyScheme.

This is the main way to instantiate new HierarchyElements.

Parameters

- **identifier** – tuple of metadata values (not keys) that define the uniqueness criteria for this element
- **atom_indices** – 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.

12.2.7 HierarchyElement

class openff.toolkit.topology.**HierarchyElement**(*scheme:* HierarchyScheme, *identifier:* *tuple[Union[str, int]]*, *atom_indices:* *Sequence[int]*)

An element in a metadata hierarchy scheme, such as a residue or chain.

__init__(*scheme:* HierarchyScheme, *identifier:* *tuple[Union[str, int]]*, *atom_indices:* *Sequence[int]*)

Create a new hierarchy element.

Parameters

- **scheme** – The scheme to which this `HierarchyElement` belongs
- **identifier** – tuple of metadata values (not keys) that define the uniqueness criteria for this element
- **atom_indices** – 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 the atom with the specified index.
`generate_unique_atom_names`()	Generate unique atom names from the element symbol and count.
`to_dict`()	Serialize this object to a basic dict of strings and lists of ints.

Attributes

`atoms`	Iterator over the atoms in this hierarchy element.
`has_unique_atom_names`	`True` if the element has unique atom names, `False` otherwise.
`n_atoms`	The number of atoms in this hierarchy element.
`parent`	The parent molecule for this hierarchy element

to_dict() → dict[str, Union[tuple[Union[str, int]], Sequence[int]]]

Serialize this object to a basic dict of strings and lists of ints.

property n_atoms: int

The number of atoms in this hierarchy element.

property atoms: Generator[Atom, None, None]

Iterator over the atoms in this hierarchy element.

atom(*index: int*) → *Atom*

Get the atom with the specified index.

property parent: Molecule | FrozenMolecule | _SimpleMolecule

The parent molecule for this hierarchy element

property has_unique_atom_names: bool

`True` if the element has unique atom names, `False` otherwise.

generate_unique_atom_names()

Generate unique atom names from the element symbol and count.

Names are generated from the elemental symbol and the number of times that element is found in the hierarchy element. 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. For example, generated atom names might begin 'C1x', 'H1x', 'O1x', 'C2x', etc.

CHAPTER

THIRTEEN

FORCE FIELD TYPING TOOLS

13.1 Force field typing engines

Engines for applying parameters to chemical systems

13.1.1 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.

ForceField

class openff.toolkit.typing.engines.smirnoff.**ForceField**(**sources, aromaticity_model: str = DEFAULT_AROMATICITY_MODEL, parameter_handler_classes: list[type[ParameterHandler]] | None = None, parameter_io_handler_classes: list[type[ParameterIOHandler]] | None = None, disable_version_check: bool = False, allow_cosmetic_attributes: bool = False, load_plugins: bool = False*)

A factory that assigns SMIRNOFF parameters to a molecular system

`ForceField` is a factory that constructs an OpenMM `System` object from a `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 object from the distributed OpenFF 2.0 ("Sage") file:

```
>>> from openff.toolkit import ForceField
>>> force_field = ForceField('openff-2.0.0.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 = force_field.create_openmm_system(topology)
```

Modify the long-range electrostatics method:

```
>>> force_field.get_parameter_handler('Electrostatics').periodic_potential = 'PME'
```

Inspect the first few vdW parameters:

```
>>> low_precedence_parameters = force_field.get_parameter_handler('vdW').parameters[0:3]
```

Retrieve the vdW parameters by SMIRKS string and manipulate it:

```
>>> from openff.toolkit import unit
>>> parameter = force_field.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):

```
>>> force_field.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 force_field.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,
... )
>>> force_field.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: str = DEFAULT_AROMATICITY_MODEL, parameter_handler_classes: list[type[ParameterHandler]] | None = None, parameter_io_handler_classes: list[type[ParameterIOHandler]] | None = None, disable_version_check: bool = False, allow_cosmetic_attributes: bool = False, load_plugins: bool = False*)

Create a new `ForceField` object from one or more SMIRNOFF parameter definition files.

Parameters

- **sources** – 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** – The aromaticity model to use. Only OEAroModel_MDL is supported.
- **parameter_handler_classes** – 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 not loaded as plugins are automatically registered.
- **parameter_io_handler_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** – 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** – Whether to retain non-spec kwargs from data sources.
- **load_plugins** – 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('openff-2.0.0.offxml')
```

Load multiple SMIRNOFF parameter sets:

```
>>> from openff.toolkit._tests.utils import get_data_file_path
>>> forcefield = ForceField('openff-2.0.0.offxml', get_data_file_path('test_
↪forcefields/tip3p.offxml'))
```

Load a parameter set from a string:

```
>>> offxml = '<SMIRNOFF version="0.2" aromaticity_model="OEAroModel_MDL"/>'
>>> forcefield = ForceField(offxml)
```

See also:

`parse_sources`

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, *[, ...])	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: str

Returns the aromaticity model for this ForceField object.

Returns

aromaticity_model – The aromaticity model for this force field.

property author: str | None

Returns the author data for this ForceField object. If not defined in any loaded files, this will be None.

Returns
author – The author data for this force field.

property date: str | None

Returns the date data for this ForceField object. If not defined in any loaded files, this will be None.

Returns
date – The date data for this force field.

register_parameter_handler(*parameter_handler: ParameterHandler*)

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 – 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: ParameterIOHandler*)

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 – 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: list[str]

Return the list of registered parameter handlers by name

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

Returns
registered_parameter_handlers

get_parameter_handler(*tagname: str, handler_kwargs: dict | None = None, allow_cosmetic_attributes: bool = False*) → *ParameterHandler*

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** – The name of the parameter to be handled.
- **handler_kwargs** – 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** – Whether to permit non-spec kwargs in smirnoff_data.

Returns
handler

Raises
KeyError – If there is no ParameterHandler for the given tagname

get_parameter_io_handler(*io_format: str*) → *ParameterIOHandler*

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 – The name of the io format to be handled.

Returns
io_handler

Raises
KeyError – If there is no ParameterIOHandler for the given tagname

deregister_parameter_handler(*handler: str* | ParameterHandler)

Deregister a parameter handler specified by tag name, class, or instance.

Parameters
handler – The handler to deregister.

parse_sources(*sources*, *allow_cosmetic_attributes: bool = True*)

Parse a SMIRNOFF force field definition.

Parameters

- **sources** – An iterable 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** – 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 end**
  of the previously-read definitions if the sections are configured with compatible attributes; otherwise, an `IncompatibleTagException` is raised.

parse_smirnoff_from_source(*source: str | IO*) → dict

Reads a SMIRNOFF data structure from a source, which can be one of many types.

Parameters

source – 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 – A representation of a SMIRNOFF-format data structure. Begins at top-level 'SMIRNOFF' key.

to_string(*io_format: str | ParameterIOHandler = 'XML', discard_cosmetic_attributes: bool = False*) → str

Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.

Parameters

- **io_format** – The serialization format to write to
- **discard_cosmetic_attributes** – Whether to discard any non-spec attributes stored in the ForceField.

Returns

forcefield_string – The string representation of the serialized force field

to_file(*filename: str, io_format: str | ParameterIOHandler | None = None, discard_cosmetic_attributes: bool = False*) → None

Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.

Parameters

- **filename** – The filename to write to
- **io_format** – The serialization format to write out. If None, will attempt to be inferred from the filename.
- **discard_cosmetic_attributes** – Whether to discard any non-spec attributes stored in the ForceField.

create_openmm_system(*topology: Topology, *, toolkit_registry: ToolkitRegistry | ToolkitWrapper | None = None, charge_from_molecules: list['Molecule'] | None = None, partial_bond_orders_from_molecules: list['Molecule'] | None = None, allow_nonintegral_charges: bool = False*) → openmm.System

Create an OpenMM System from this ForceField and a Topology.

Note that most force fields specify their own partial charges, and any partial charges defined on the `Molecule` objects in the topology are ignored. To use custom partial charges, see the `charge_from_molecules` argument.

Parameters

- **topology** – The `Topology` which is to be parameterized with this `ForceField`.
- **toolkit_registry** – The toolkit registry to use for parametrization (eg, for calculating partial charges and partial bond orders)
- **charge_from_molecules** – Take partial charges from the input topology rather than calculating them. This may be useful for avoiding recalculating charges, but take care to ensure that your charges are appropriate for the force field.
- **partial_bond_orders_from_molecules** – Take partial bond orders from the input topology rather than calculating them. This may be useful for avoiding recalculating PBOs, but take to ensure that they are appropriate for the force field.
- **allow_nonintegral_charges** – Allow charges that do not sum to an integer.

create_interchange(*topology:* Topology, *toolkit_registry:* ToolkitRegistry | ToolkitWrapper | *None* = *None*, *charge_from_molecules:* *list['Molecule']* | *None* = *None*, *partial_bond_orders_from_molecules:* *list['Molecule']* | *None* = *None*, *allow_nonintegral_charges:* *bool* = *False*) → Interchange

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** – The topology to create this *Interchange* object from.
- **toolkit_registry** – The toolkit registry to use for parametrization (eg, for calculating partial charges and partial bond orders)
- **charge_from_molecules** – Take charges from the input topology rather than calculating them. This may be useful for avoiding recalculating charges, but take care to ensure that your charges are appropriate for the force field.
- **partial_bond_orders_from_molecules** – Take partial bond orders from the input topology rather than calculating them. This may be useful for avoiding recalculating PBOs, but take to ensure that they are appropriate for the force field.
- **allow_nonintegral_charges** – Allow charges that do not sum to an integer.

Returns

interchange – An *Interchange* object resulting from applying this *ForceField* to a *Topology*.

label_molecules(*topology:* Topology) → list[dict[str, 'ValenceDict']]

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 – A Topology object containing one or more unique molecules to be labeled

Returns

molecule_labels – 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), ... ]`.

get_partial_charges(*molecule:* Molecule, ***kwargs:* *Any*) → Quantity

Generate the partial charges for the given molecule in this force field.

Parameters

- **molecule** – The `Molecule` corresponding to the system to be parameterized
- **toolkit_registry** – The toolkit registry to use for operations like conformer generation and partial charge assignment.

Returns

charges – 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('openff-2.0.0.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 = force_field.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.

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 – 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 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	Store virtual site parameters (geometry and electrostatics) and vdW interactions.

ParameterType

class openff.toolkit.typing.engines.smirnoff.parameters.**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** – The SMIRKS pattern that this parameter matches.
- **id** – An optional identifier for the parameter.

- **parent_id** – 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, _ELEMENT_NAME is used to describe the SMIRNOFF parameter being defined, and is used during automatic serialization/deserialization into a dict.

```
>>> class MyBondParameter(ParameterType):
...     _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(1.01, 'angstrom')>
>>> my_par.k = 3.0 * unit.gram
Traceback (most recent call last):
...
openff.toolkit.utils.exceptions.IncompatibleUnitError:
k=3.0 gram 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):
...     _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")
```

(continues on next page)

(continued from previous page)

```
...         return value
...
>>> my_par = MyParameterType(smirks='[*:1]', attr_all_to_float='3.0', attr_int_to_
↪float=1)
>>> my_par.attr_optional
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):
...     _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** – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** – 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns

is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns

smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

ConstraintType

class openff.toolkit.typing.engines.smirnoff.parameters.**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** – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** – 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns

is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns

smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

BondType

class openff.toolkit.typing.engines.smirnoff.parameters.**BondType**(***kwargs*)

A SMIRNOFF bond type

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

__init__(***kwargs*)

Create a ParameterType.

Parameters

- **smirks** – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** – 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

AngleType

class openff.toolkit.typing.engines.smirnoff.parameters.**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** – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** – 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

ProperTorsionType

class openff.toolkit.typing.engines.smirnoff.parameters.**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** – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** – 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

ImproperTorsionType

class openff.toolkit.typing.engines.smirnoff.parameters.**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** – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** – 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

vdWType

class openff.toolkit.typing.engines.smirnoff.parameters.**vdWType**(***kwargs*)

A SMIRNOFF vdWForce type.

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

__init__(***kwargs*)

Create a ParameterType.

Parameters

- **smirks** – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** – 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – 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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

LibraryChargeType

class openff.toolkit.typing.engines.smirnoff.parameters.**LibraryChargeType**(***kwargs*)

A SMIRNOFF Library Charge type.

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

__init__(***kwargs*)

Create a ParameterType.

Parameters

- **smirks** – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** – 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

classmethod from_molecule(*molecule:* Molecule)

Construct a LibraryChargeType from a molecule with existing partial charges.

Parameters
molecule – The molecule to create the LibraryChargeType from. The molecule must have partial charges.

Returns
library_charge_type – A LibraryChargeType that is expected to match this molecule and its partial charges.

Raises
MissingPartialChargesError –

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

GBSAType

class openff.toolkit.typing.engines.smirnoff.parameters.**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** – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** – 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns

is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns

smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

ChargeIncrementType

class openff.toolkit.typing.engines.smirnoff.parameters.**ChargeIncrementType**(***kwargs*)

A SMIRNOFF bond charge correction type.

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

__init__(***kwargs*)

Create a ParameterType.

Parameters

- **smirks** – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** – 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

VirtualSiteType

class openff.toolkit.typing.engines.smirnoff.parameters.**VirtualSiteType**(***kwargs*)

Store virtual site parameters (geometry and electrostatics) and vdW interactions.

__init__(***kwargs*)

Create a ParameterType.

Parameters

- **smirks** – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** – 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns

is_cosmetic – 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 – 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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns

smirnoff_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 .

ParameterList

class openff.toolkit.typing.engines.smirnoff.parameters.**ParameterList**(*input_parameter_list: list['ParameterType'] | None = None*)

Parameter list that also supports accessing items by SMARTS string.

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

__init__(*input_parameter_list: list['ParameterType'] | None = None*)

Initialize a new ParameterList, optionally providing a list of ParameterType objects to initially populate it.

Parameters

input_parameter_list – 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 –

extend(*other*)

Add a ParameterList object to the end of the ParameterList

Parameters
other –

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 – The parameter or SMIRKS to look up in this ParameterList

Returns
index – 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** – The numerical position to insert the parameter at
- **parameter** – 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 – 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.

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** – 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** – 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)	Deprecated since version 0.11.0.
`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 a dict, 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 – 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 – The kwargs that would be used to construct

Raises
IncompatibleParameterError if handler_kwargs are incompatible with existing parameters. –

add_parameter(*parameter_kwargs: dict | None = None, parameter: ParameterType | None = None, after: str | None = None, before: str | None = None, allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler
- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

find_matches(*entity:* Topology, *unique:* *bool* = *False*) → *ValenceDict*

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** – Topology to search.
- **unique** – 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 – `matches[atom_indices]` is the `ParameterType` object matching the tuple of atom indices in `entity`.

create_force(**args*, ***kwarsg*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

to_dict(*discard_cosmetic_attributes:* *bool* = *False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data – 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

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** – 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** – 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)	Deprecated since version 0.11.0.
`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 a dict, 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check
Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_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 – 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** – Name of the attribute to define for this object.
- **attr_value** – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs: dict | None = None, parameter: ParameterType | None = None, after: str | None = None, before: str | None = None, allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler
- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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'}
↪)
```

(continues on next page)

(continued from previous page)

```
>>> [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 – The attribute name to check

Returns

is_cosmetic – 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 – The kwargs that would be used to construct

Raises

IncompatibleParameterError if handler_kwargs are incompatible with existing parameters. –

create_force(**args*, ***kwarsg*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

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 – Name of the cosmetic attribute to delete.

find_matches(*entity: Topology*, *unique: bool = False*) → *ValenceDict*

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** – Topology to search.
- **unique** – 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 – `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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns
params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

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: bool = False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters
discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns
smirnoff_data – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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** – 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** – 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)	Deprecated since version 0.11.0.
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 a dict, 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns

is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler: BondHandler*)

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 – 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 – 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** – Name of the attribute to define for this object.
- **attr_value** – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs: dict | None = None, parameter: ParameterType | None = None, after: str | None = None, before: str | None = None, allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler
- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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 – The attribute name to check

Returns

is_cosmetic – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

create_force(**args*, ***kwarsg*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

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 – Name of the cosmetic attribute to delete.

find_matches(*entity: Topology*, *unique: bool = False*) → *ValenceDict*

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** – Topology to search.
- **unique** – 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 – `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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

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: bool = False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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** – 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** – 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)	Deprecated since version 0.11.0.
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 a dict, 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False*, *duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler:* AngleHandler)

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 – 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 – 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** – Name of the attribute to define for this object.
- **attr_value** – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs: dict | None = None, parameter:* ParameterType *| None = None, after: str | None = None, before: str | None = None, allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler
- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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 – The attribute name to check

Returns

is_cosmetic – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

create_force(**args, **kwarsg*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

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 – Name of the cosmetic attribute to delete.

find_matches(*entity: Topology, unique: bool = False*) → *ValenceDict*

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** – Topology to search.
- **unique** – 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 – `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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

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: bool = False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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** – 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** – 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)	Deprecated since version 0.11.0.
`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 a dict, 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler: ProperTorsionHandler*)

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 – 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 – 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** – Name of the attribute to define for this object.
- **attr_value** – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs: dict | None = None, parameter: ParameterType | None = None, after: str | None = None, before: str | None = None, allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler
- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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 – The attribute name to check

Returns

is_cosmetic – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

create_force(**args, **kwarsg*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

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 – Name of the cosmetic attribute to delete.

find_matches(*entity: Topology, unique: bool = False*) → *ValenceDict*

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** – Topology to search.
- **unique** – 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 – `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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns
params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

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: bool = False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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** – 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** – 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)	Deprecated since version 0.11.0.
`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 a dict, 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler: ImproperTorsionHandler*)

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 – 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 to search.

Returns

matches – `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 – 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** – Name of the attribute to define for this object.
- **attr_value** – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs: dict | None = None, parameter: ParameterType | None = None, after: str | None = None, before: str | None = None, allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler
- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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 – The attribute name to check

Returns

is_cosmetic – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

create_force(**args, **kwarsg*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

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 – 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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

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: bool = False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns
smirnoff_data – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

vdWHandler

class `openff.toolkit.typing.engines.smirnoff.parameters.`**`vdWHandler`**(***kwargs*)

Handle SMIRNOFF <vdW> 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`** – 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`** – 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)	Deprecated since version 0.11.0.
`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 a dict, 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.
nonperiodic_method	
parameters	The ParameterList that holds this ParameterHandler's parameter objects
periodic_method	
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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns

smirnoff_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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

check_handler_compatibility(*other_handler:* vdWHandler)

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 – 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 – 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** – Name of the attribute to define for this object.
- **attr_value** – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs: dict | None = None, parameter: ParameterType | None = None, after: str | None = None, before: str | None = None, allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler
- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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 – The attribute name to check

Returns

is_cosmetic – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

create_force(**args*, ***kwarsg*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

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 – Name of the cosmetic attribute to delete.

find_matches(*entity: Topology*, *unique: bool = False*) → *ValenceDict*

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** – Topology to search.
- **unique** – 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 – `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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

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: bool = False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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** – 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** – 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)	Deprecated since version 0.11.0.
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 a dict, 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:* ElectrostaticsHandler)

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 – 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 – 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** – Name of the attribute to define for this object.
- **attr_value** – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs: dict | None = None, parameter: ParameterType | None = None, after: str | None = None, before: str | None = None, allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler
- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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 – The attribute name to check

Returns

is_cosmetic – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

create_force(**args*, ***kwarsg*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

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 – Name of the cosmetic attribute to delete.

find_matches(*entity: Topology*, *unique: bool = False*) → *ValenceDict*

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** – Topology to search.
- **unique** – 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 – `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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns
params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

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: bool = False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters
discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns
smirnoff_data – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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** – 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** – 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)	Deprecated since version 0.11.0.
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 a dict, 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 – The molecule to create the LibraryChargeType from. The molecule must have partial charges.

Returns

library_charge_type – A LibraryChargeType that is expected to match this molecule and its partial charges.

Raises

MissingPartialChargesError –

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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns

is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns

smirnoff_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 to search.

Returns

matches – `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 – 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** – Name of the attribute to define for this object.
- **attr_value** – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs: dict | None = None, parameter: ParameterType | None = None, after: str | None = None, before: str | None = None, allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler
- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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 – The attribute name to check

Returns

is_cosmetic – 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 – The kwargs that would be used to construct

Raises

IncompatibleParameterError if handler_kwargs are incompatible with existing parameters. –

create_force(**args*, ***kwarsg*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

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 – 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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

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: bool = False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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** – 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** – 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)	Deprecated since version 0.11.0.
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 a dict, 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:* ToolkitAM1BCCHandler, *assume_missing_is_default: bool = 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** – The handler to compare to.

- **assume_missing_is_default** –

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 – 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** – Name of the attribute to define for this object.
- **attr_value** – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs: dict | None = None, parameter: ParameterType | None = None, after: str | None = None, before: str | None = None, allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler
- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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 – The attribute name to check

Returns

is_cosmetic – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

create_force(**args, **kwarsg*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

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 – Name of the cosmetic attribute to delete.

find_matches(*entity: Topology, unique: bool = False*) → *ValenceDict*

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** – Topology to search.
- **unique** – 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 – `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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns
params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

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: bool = False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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** – 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** – 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)	Deprecated since version 0.11.0.
`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 a dict, 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False*, *duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler:* GBSAHandler)

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 – 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 – 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** – Name of the attribute to define for this object.
- **attr_value** – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs: dict | None = None, parameter:* ParameterType *| None = None, after: str | None = None, before: str | None = None, allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler
- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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 – The attribute name to check

Returns

is_cosmetic – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

create_force(**args, **kwarsg*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

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 – Name of the cosmetic attribute to delete.

find_matches(*entity: Topology, unique: bool = False*) → *ValenceDict*

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** – Topology to search.
- **unique** – 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 – `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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

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: bool = False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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** – 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** – 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)	Deprecated since version 0.11.0.
`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 a dict, 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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes: bool = False, duplicate_attributes: list[str] | None = None*) → dict

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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns
smirnoff_dict – The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler: ChargeIncrementModelHandler, assume_missing_is_default: bool = 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** – The handler to compare to.
- **assume_missing_is_default** –

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 to search.

Returns
matches – `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 – 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** – Name of the attribute to define for this object.
- **attr_value** – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs: dict | None = None, parameter: ParameterType | None = None, after: str | None = None, before: str | None = None, allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler
- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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 – The attribute name to check

Returns

is_cosmetic – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

create_force(**args, **kwargs*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

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 – 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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

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: bool = False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns
smirnoff_data – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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** – 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** – 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)	Deprecated since version 0.11.0.
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 a dict, 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*)

Store virtual site parameters (geometry and electrostatics) and vdW interactions.

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** – Name of the attribute to define for this object.
- **attr_value** – 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 – The attribute name to check

Returns
is_cosmetic – 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 – 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** – Whether to discard non-spec attributes of this object
- **duplicate_attributes** – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns

smirnoff_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.

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns

handler_name – 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** – Name of the attribute to define for this object.
- **attr_value** – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs: dict | None = None*, *parameter: ParameterType | None = None*, *after: str | None = None*, *before: str | None = None*, *allow_duplicate_smirks: bool = False*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** – A ParameterType to add to the ParameterHandler

- **after** – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **allow_duplicate_smirks** – If False, a DuplicateParameterError will be raised if the parameter being added has a SMIRKS that already appears in another parameter owned by this ParameterHandler.
- **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 openff.toolkit 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 – The attribute name to check

Returns
is_cosmetic – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

create_force(**args*, ***kwarsg*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0, no longer has any functionality, and will soon be removed. Use the OpenFF Interchange package instead.

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 – Name of the cosmetic attribute to delete.

find_matches(*entity:* Topology, *unique:* *bool* = *False*) → *ValenceDict*

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** – Topology to search.
- **unique** – 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 – `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 – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]*", "id": "t105"})

Returns
params – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openff.toolkit 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 angstrom  k: 10.0 kilocalorie /␣
↪angstrom ** 2 / mole  >]
```

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: bool = False*) → dict

Convert this ParameterHandler to a dict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

check_handler_compatibility(*other_handler: VirtualSiteHandler*)

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 – 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.

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`** – The path to the file to write to.
- **`smirnoff_data`** – 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` – A dictionary structured in compliance with the SMIRNOFF spec

Returns
str

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 – 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: str*) → dict

Parse a SMIRNOFF force field definition in XML format.

A `SMIRNOFFParseError` is raised if the XML cannot be processed.

Parameters
data – 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** – The path to the file to be written. The *.offxml* or *.xml* file extension must be present.
- **smirnoff_data** – A dict structured in compliance with the SMIRNOFF data spec.

to_string(*smirnoff_data: dict*) → str

Write the current force field parameter set to an XML string.

Parameters

smirnoff_data – A dictionary structured in compliance with the SMIRNOFF spec

Returns

serialized_forcefield – 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.

ParameterAttribute

class openff.toolkit.typing.engines.smirnoff.parameters.**ParameterAttribute**(*default: Any = UNDEFINED, unit: Unit | None = None, converter: Callable | None = None, docstring: str = ''*)

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** – When specified, the descriptor makes this attribute optional by attaching a default value to it.
- **unit** – When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.
- **converter** – 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'. Did you mean:
↪'attr_required'?
```

The attribute allow automatic conversion and validation of units.

```
>>> from openff.toolkit 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(1.0, 'nanometer')>
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openff.toolkit.utils.exceptions.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
```

(continues on next page)

(continued from previous page)

```
...         # 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: Any = UNDEFINED, unit: Unit | None = None, converter: Callable | None = None, docstring: str = ''*)

Methods

`__init__`([default, unit, converter, docstring])	
`converter`(converter)	Create a new ParameterAttribute with an associated converter.

Attributes

`name`

UNDEFINED

Marker type for an undeclared default parameter.

alias of `_UNDEFINED`

converter(*converter*)

Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

IndexedParameterAttribute

class openff.toolkit.typing.engines.smirnoff.parameters.**IndexedParameterAttribute**(*default: Any = UNDEFINED, unit: Unit | None = None, converter: Callable | None = None, docstring: str = ''*)

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** – When specified, the descriptor makes this attribute optional by attaching a default value to it.
- **unit** – When specified, only sequences of quantities with compatible units are allowed to be set.
- **converter** – 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 openff.toolkit 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(1, '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: Any = UNDEFINED, unit: Unit | None = None, converter: Callable | None = None, docstring: str = ''*)

Methods

`__init__`([default, unit, converter, docstring])	
`converter`(converter)	Create a new ParameterAttribute with an associated converter.

Attributes

`name`

UNDEFINED

alias of `_UNDEFINED`

converter(*converter*)

Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

MappedParameterAttribute

class openff.toolkit.typing.engines.smirnoff.parameters.**MappedParameterAttribute**(*default: Any = UNDEFINED, unit: Unit | None = None, converter: Callable | None = None, docstring: str = ''*)

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** – When specified, the descriptor makes this attribute optional by attaching a default value to it.
- **unit** – When specified, only sequences of mappings where values are quantities with compatible units are allowed to be set.
- **converter** – 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 openff.toolkit 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(1.5, '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(1.42, 'angstrom')>
```

__init__(*default: Any = UNDEFINED, unit: Unit | None = None, converter: Callable | None = None, docstring: str = ''*)

Methods

`__init__`([default, unit, converter, docstring])	
`converter`(converter)	Create a new ParameterAttribute with an associated converter.

Attributes

`name`

UNDEFINED

alias of `_UNDEFINED`

converter(*converter*)

Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

IndexedMappedParameterAttribute

class openff.toolkit.typing.engines.smirnoff.parameters.**IndexedMappedParameterAttribute**(*default: Any = UNDEFINED, unit: Unit | None = None, converter: Callable | None = None, docstring: str = ''*)

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** – When specified, the descriptor makes this attribute optional by attaching a default value to it.
- **unit** – When specified, only sequences of mappings where values are quantities with compatible units are allowed to be set.
- **converter** – 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 openff.toolkit 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(1, '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: Any = UNDEFINED, unit: Unit | None = None, converter: Callable | None = None, docstring: str = ''*)

Methods

`__init__`([default, unit, converter, docstring])	
`converter`(converter)	Create a new ParameterAttribute with an associated converter.

Attributes

`name`

UNDEFINED

alias of `_UNDEFINED`

converter(*converter*)

Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

CHAPTER

FOURTEEN

UTILITIES

14.1 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.

To temporarily change the state of `GLOBAL_TOOLKIT_REGISTRY`, we provide the `toolkit_registry_manager` context manager.

```
>>> from openff.toolkit.utils.toolkits import RDKitToolkitWrapper, AmberToolsToolkitWrapper,␣
↪GLOBAL_TOOLKIT_REGISTRY
>>> from openff.toolkit.utils import toolkit_registry_manager
>>> print(len(GLOBAL_TOOLKIT_REGISTRY.registered_toolkits))
4
>>> with toolkit_registry_manager(ToolkitRegistry([RDKitToolkitWrapper(),␣
↪AmberToolsToolkitWrapper()])):
...     print(len(GLOBAL_TOOLKIT_REGISTRY.registered_toolkits))
2
```

To remove `ToolkitWrappers` permanently from a `ToolkitRegistry`, the `deregister_toolkit` method can be used:

```
>>> from openff.toolkit.utils.toolkits import OpenEyeToolkitWrapper, BuiltInToolkitWrapper
>>> from openff.toolkit.utils.toolkits import GLOBAL_TOOLKIT_REGISTRY as toolkit_registry
>>> print(len(toolkit_registry.registered_toolkits))
4
>>> toolkit_registry.deregister_toolkit(RDKitToolkitWrapper)
>>> print(len(toolkit_registry.registered_toolkits))
3
>>> toolkit_registry.register_toolkit(RDKitToolkitWrapper)
>>> print(len(toolkit_registry.registered_toolkits))
4
```

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
`NAGLToolkitWrapper`	NAGL toolkit wrapper for applying partial charges with a GCN model.
`BuiltInToolkitWrapper`	Built-in ToolkitWrapper for very basic functionality.

14.1.1 ToolkitRegistry

class openff.toolkit.utils.toolkits.**ToolkitRegistry**(*toolkit_precedence: list[type[ToolkitWrapper]] | None = None, exception_if_unavailable: bool = True, _register_imported_toolkit_wrappers: bool = 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: list[type[ToolkitWrapper]] | None = None, exception_if_unavailable: bool = True, _register_imported_toolkit_wrappers: bool = False*)

Create an empty toolkit registry.

Parameters

- **toolkit_precedence** – List of toolkit wrapper classes, in order of desired precedence when performing molecule operations. If None, no toolkits will be registered.
- **exception_if_unavailable** – If True, an exception will be raised if the toolkit is unavailable

- **_register_imported_toolkit_wrappers** – 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[ToolkitWrapper]

List registered toolkits.

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

Returns
toolkits

property registered_toolkit_versions: dict[str, str]

Return a dict containing the version of each registered toolkit.

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

Returns
toolkit_versions – A dictionary mapping names and versions of wrapped toolkits

register_toolkit(*toolkit_wrapper: ToolkitWrapper | type[ToolkitWrapper], exception_if_unavailable: bool = True*)

Register the provided toolkit wrapper class, instantiating an object of it.

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

Parameters

- **toolkit_wrapper** – The toolkit wrapper to register or its class.
- **exception_if_unavailable** – If True, an exception will be raised if the toolkit is unavailable

deregister_toolkit(*toolkit_wrapper:* ToolkitWrapper)

Remove a ToolkitWrapper from the list of toolkits in this ToolkitRegistry

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

Parameters

toolkit_wrapper – 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:* ToolkitWrapper)

Append a ToolkitWrapper onto the list of toolkits in this ToolkitRegistry

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

Parameters

toolkit_wrapper – 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: str*) → Callable

Resolve the requested method name by checking all registered toolkits in order of precedence for one that provides the requested method.

Parameters

method_name – 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: str, *args, raise_exception_types: list[type[Exception]] | None = 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** – The name of the method to execute
- **raise_exception_types** – 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)
```

14.1.2 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

supported_charge_methods	
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 – 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[str]

List of file formats that this toolkit can write.

classmethod is_available()

Check whether the corresponding toolkit can be imported

Returns

is_installed – 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 – 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** – The file to read the molecule from
- **file_format** – 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** – If false, raises an exception if any molecules contain undefined stereochemistry.
- **_cls** – Molecule constructor

Returns

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** – The file-like object to read the molecule from
- **file_format** – 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** – If false, raises an exception if any molecules contain undefined stereochemistry. If false, the function skips loading the molecule.
- **_cls** – Molecule constructor

Returns

molecules – a list of Molecule objects is returned.

14.1.3 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.
`enumerate_protomers`(molecule[, max_states])	Enumerate the formal charges of a molecule to generate different protomers.
`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, ...])	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 OpenEye OEChem Toolkit.
`to_inchikey`(molecule[, fixed_hydrogens])	Create an InChIKey for the molecule using the OpenEye OEChem 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

SUPPORTED_CHARGE_METHODS	
supported_charge_methods	
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() → bool

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 – `True` if all required OpenEye tools are installed and licensed, `False` otherwise

from_object(*obj*, *allow_undefined_stereo: bool = False*, *_cls=None*) → *FrozenMolecule*

Convert an OEMol (or OEMol-derived object) into an openff.toolkit.topology.molecule

Parameters

- **obj** – An object to by type-checked.
- **allow_undefined_stereo** – 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** – 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: str | Path*, *file_format: str*, *allow_undefined_stereo: bool = False*, *_cls=None*) → list['FrozenMolecule']

Return an openff.toolkit.topology.Molecule from a file using this toolkit.

Parameters

- **file_path** – The file to read the molecule from
- **file_format** – 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** – If false, raises an exception if oemol contains undefined stereochemistry.

- **_cls** – Molecule constructor

Returns
molecules – 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: str*, *allow_undefined_stereo: bool = False*, *_cls=None*) → list['Molecule']

Return an openff.toolkit.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.

Parameters

- **file_obj** – The file-like object to read the molecule from
- **file_format** – 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** – If false, raises an exception if oemol contains undefined stereochemistry.
- **_cls** – Molecule constructor

Returns
molecules – 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: Molecule*, *file_obj*, *file_format: str*)

Writes an OpenFF Molecule to a file-like object

Parameters

- **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: Molecule*, *file_path: str | Path*, *file_format: str*)

Writes an OpenFF Molecule to a file-like object

Parameters

- **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:* FrozenMolecule, *max_states:* *int* = *0*) → list['FrozenMolecule']

Enumerate the formal charges of a molecule to generate different protomers. Note that, in cases where the input molecule has an uncommon protonation state (for example `[NH2-]`), the input molecule may not be included in the output.

Parameters

- **molecule** – The molecule whose state we should enumerate
- **max_states** – The maximum number of protomer states to be returned. If 0, the default, attempt to return all protomers.

Returns

molecules – A list of the protomers of the input molecules, including the input molecule if found by Quacpac and not pruned by *max_states*.

enumerate_stereoisomers(*molecule:* FrozenMolecule, *undefined_only:* *bool* = *False*, *max_isomers:* *int* = *20*, *rationalise:* *bool* = *True*) → list['FrozenMolecule']

Enumerate the stereocenters and bonds of the current molecule.

Parameters

- **molecule** – The molecule whose state we should enumerate
- **undefined_only** – If we should enumerate all stereocenters and bonds or only those with undefined stereochemistry
- **max_isomers** – The maximum amount of molecules that should be returned
- **rationalise** – If we should try to build and rationalise the molecule to ensure it can exist

Returns

molecules – A list of openff.toolkit.topology.Molecule instances

enumerate_tautomers(*molecule:* FrozenMolecule, *max_states:* *int* = *20*) → list['FrozenMolecule']

Enumerate the possible tautomers of the current molecule

Parameters

- **molecule** – The molecule whose state we should enumerate
- **max_states** – The maximum amount of molecules that should be returned

Returns

molecules – A list of openff.toolkit.topology.Molecule instances excluding the input molecule.

static from_openeye(*oemol*, *allow_undefined_stereo:* *bool* = *False*, *_cls=None*) → *FrozenMolecule*

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** – An OpenEye molecule
- **allow_undefined_stereo** – If false, raises an exception if oemol contains undefined stereochemistry.
- **_cls** – Molecule constructor

Returns
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:* FrozenMolecule, *aromaticity_model:* *str* = *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** – The molecule to convert to an OEMol
- **aromaticity_model** – The aromaticity model to use

Returns
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 – The molecule containing the atom of interest

Returns
is_in_ring – 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 – The molecule containing the atom of interest

Returns
is_in_ring – Whether or not the bond of index *bond_index* is in a ring

Raises
NotAttachedToMoleculeError –

to_smiles(*molecule:* Molecule, *isomeric:* bool = *True*, *explicit_hydrogens:* bool = *True*, *mapped:* bool = *False*) → str

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** – The molecule to convert into a SMILES.
- **isomeric** – return an isomeric smiles
- **explicit_hydrogens** – return a smiles string containing all hydrogens explicitly
- **mapped** – 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 – The SMILES of the input molecule.

to_inchi(*molecule:* Molecule, *fixed_hydrogens:* *bool* = *False*) → str

Create an InChI string for the molecule using the OpenEye OEChem 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** – The molecule to convert into a SMILES.
- **fixed_hydrogens** – 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:* Molecule, *fixed_hydrogens:* *bool* = *False*) → str

Create an InChIKey for the molecule using the OpenEye OEChem 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** – The molecule to convert into a SMILES.
- **fixed_hydrogens** – 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 – The InChIKey representation of the molecule.

to_iupac(*molecule:* Molecule) → str

Generate IUPAC name from Molecule

Parameters
molecule – The molecule to convert into a SMILES.

Returns
iupac_name – 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:* FrozenMolecule) → *FrozenMolecule*

Canonical order the atoms in the molecule using the OpenEye toolkit.

Parameters

- **molecule** – The input molecule

  Returns

- **-------** –

- **molecule** – The input molecule, with canonically-indexed atoms and bonds.

from_smiles(*smiles: str, hydrogens_are_explicit: bool = False, allow_undefined_stereo: bool = False, _cls=None, name: str = ''*) → *FrozenMolecule*

Create a Molecule from a SMILES string using the OpenEye toolkit.

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

Parameters

- **smiles** – The SMILES string to turn into a molecule
- **hydrogens_are_explicit** – If False, OE will perform hydrogen addition using OEAddExplicitHydrogens
- **allow_undefined_stereo** – 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** – Molecule constructor
- **name** – An optional name for the output molecule

Returns
molecule – An OpenFF style molecule.

Raises
RadicalsNotSupportedError –

from_inchi(*inchi: str, allow_undefined_stereo: bool = False, _cls=None, name: str = ''*) → *FrozenMolecule*

Construct a Molecule from a InChI representation

Parameters

- **inchi** – The InChI representation of the molecule.
- **allow_undefined_stereo** – 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** – Molecule constructor
- **name** – An optional name for the output molecule

Returns
molecule

from_iupac(*iupac_name: str, allow_undefined_stereo: bool = False, _cls=None, **kwargs*) → *FrozenMolecule*

Construct a Molecule from an IUPAC name

Parameters

- **iupac_name** – The IUPAC or common name of the molecule.

- **allow_undefined_stereo** – 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** – Molecule constructor

Returns
molecule

generate_conformers(*molecule: Molecule, n_conformers: int = 1, rms_cutoff: Quantity | None = None, clear_existing: bool = True, make_carboxylic_acids_cis: bool = False*)

Generate molecule conformers using OpenEye Omega.

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

Parameters

- **molecule** – The molecule to generate conformers for.
- **n_conformers** – The maximum number of conformers to generate.
- **rms_cutoff** – 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** – Whether to overwrite existing conformers for the molecule
- **make_carboxylic_acids_cis** – 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: Molecule, partial_charge_method: str | None = None, use_conformers: list[Quantity] | None = None, strict_n_conformers: bool = False, normalize_partial_charges: bool = 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 for which partial charges are to be computed
- **partial_charge_method** – The charge model to use. One of ['amberff94', 'mmff', 'mmff94', 'am1-mulliken', 'am1bcc', 'am1bccnosymspt', 'am1bccelf10', 'gasteiger'] If None, 'am1-mulliken' will be used.
- **use_conformers** – 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** – 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** – 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** – 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** –

assign_fractional_bond_orders(*molecule: Molecule, bond_order_model: str | None = None, use_conformers: list[Quantity] | None = 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** – The molecule to assign wiberg bond orders to
- **bond_order_model** – 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** – 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** – Molecule constructor

get_tagged_smarts_connectivity(*smarts: str*) → tuple[tuple[tuple[Any, ...], ...], ...]

Returns a tuple of tuples indicating connectivity between tagged atoms in a SMARTS string. Does not return bond order.

Parameters

smarts – The tagged SMARTS to analyze

Returns

- *unique_tags* – A sorted tuple of all unique tagged atom map indices.
- *tagged_atom_connectivity* – 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: Molecule, smarts: str, aromaticity_model=DEFAULT_AROMATICITY_MODEL, unique=False*) → list[tuple[int, ...]]

Find all SMARTS matches for the specified molecule, using the specified aromaticity model.

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

Parameters

- **molecule** – The molecule for which all specified SMARTS matches are to be located
- **smarts** – SMARTS string with optional SMIRKS-style atom tagging
- **aromaticity_model** – The aromaticity model to use. Only OEAroModel_MDL is supported.
- **unique** – If True, only return unique matches. If False, return all matches.
- `:(.. 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[str]

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 – 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 – The version of the wrapped toolkit

14.1.4 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, ...])	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

SUPPORTED_CHARGE_METHODS	
supported_charge_methods	
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[str]

List of file formats that this toolkit can write.

classmethod is_available() → bool

Check whether the RDKit toolkit can be imported

Returns
is_installed – True if RDKit is installed, False otherwise.

from_object(*obj*, *allow_undefined_stereo: bool = 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** – An object to be type-checked and converted into a Molecule, if possible.
- **allow_undefined_stereo** – 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** – 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: str*, *smiles: str*, *allow_undefined_stereo: bool = False*, *_cls=None*, *name: str = ''*)

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** – PDB file path
- **smiles** – a valid smiles string for the pdb, used for stereochemistry, formal charges, and bond order
- **allow_undefined_stereo** – If false, raises an exception if SMILES contains undefined stereochemistry.
- **_cls** – Molecule constructor
- **name** – An optional name for the output molecule

Returns
molecule – An OFFMol instance with ordering the same as used in the PDB file.

Raises
InvalidConformerError –

from_file(*file_path: str, file_format: str, allow_undefined_stereo: bool = False, _cls=None*)

Create an openff.toolkit.topology.Molecule from a file using this toolkit.

Parameters

- **file_path** – The file to read the molecule from
- **file_format** – 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** – If false, raises an exception if RDMol contains undefined stereochemistry.
- **_cls** – Molecule constructor

Returns
molecules – a list of Molecule objects is returned.

from_file_obj(*file_obj, file_format: str, allow_undefined_stereo: bool = 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** – The file-like object to read the molecule from
- **file_format** – 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** – If false, raises an exception if RDMol contains undefined stereochemistry.
- **_cls** – Molecule constructor

Returns
molecules – a list of Molecule objects is returned.

to_file_obj(*molecule:* Molecule, *file_obj*, *file_format:* *str*)

Writes an OpenFF Molecule to a file-like object

Parameters

- **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:* Molecule, *file_path:* *str*, *file_format:* *str*)

Writes an OpenFF Molecule to a file-like object

Parameters

- **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:* Molecule, *undefined_only:* *bool* = *False*, *max_isomers:* *int* = *20*, *rationalise:* *bool* = *True*) → list['Molecule']

Enumerate the stereocenters and bonds of the current molecule.

Parameters

- **molecule** – The molecule whose state we should enumerate
- **undefined_only** – If we should enumerate all stereocenters and bonds or only those with undefined stereochemistry
- **max_isomers** – The maximum amount of molecules that should be returned
- **rationalise** – If we should try to build and rationalise the molecule to ensure it can exist

Returns

molecules – A list of openff.toolkit.topology.Molecule instances

enumerate_tautomers(*molecule:* Molecule, *max_states:* *int* = *20*) → list['Molecule']

Enumerate the possible tautomers of the current molecule.

Parameters

- **molecule** – The molecule whose state we should enumerate
- **max_states** – The maximum amount of molecules that should be returned

Returns

molecules – A list of openff.toolkit.topology.Molecule instances not including the input molecule.

canonical_order_atoms(*molecule:* Molecule) → *Molecule*

Canonical order the atoms in the molecule using the RDKit.

Parameters

- **molecule** – The input molecule

  Returns

- **-------** –
- **molecule** – The input molecule, with canonically-indexed atoms and bonds.

to_smiles(*molecule: Molecule, isomeric: bool = True, explicit_hydrogens: bool = True, mapped: bool = 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** – The molecule to convert into a SMILES.
- **isomeric** – return an isomeric smiles
- **explicit_hydrogens** – return a smiles string containing all hydrogens explicitly
- **mapped** – 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 – The SMILES of the input molecule.

from_smiles(*smiles: str, hydrogens_are_explicit: bool = False, allow_undefined_stereo: bool = False, _cls=None, name: str = ''*)

Create a Molecule from a SMILES string using the RDKit toolkit.

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

Parameters

- **smiles** – The SMILES string to turn into a molecule
- **hydrogens_are_explicit** – If False, RDKit will perform hydrogen addition using Chem.AddHs
- **allow_undefined_stereo** – 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** – Molecule constructor
- **name** – An optional name to pass to the _cls constructor

Returns

molecule – An OpenFF style molecule.

Raises

RadicalsNotSupportedError –

from_inchi(*inchi: str, allow_undefined_stereo: bool = False, _cls=None, name: str = ''*)

Construct a Molecule from a InChI representation

Parameters

- **inchi** – The InChI representation of the molecule.
- **allow_undefined_stereo** – 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** – Molecule constructor

Returns
molecule

generate_conformers(*molecule: Molecule, n_conformers: int = 1, rms_cutoff: Quantity | None = None, clear_existing: bool = True, _cls=None, make_carboxylic_acids_cis: bool = False*)

Generate molecule conformers using RDKit.

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

Parameters

- **molecule** – The molecule to generate conformers for.
- **n_conformers** – Maximum number of conformers to generate.
- **rms_cutoff** – 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** – Whether to overwrite existing conformers for the molecule.
- **_cls** – Molecule constructor
- **make_carboxylic_acids_cis** – 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: Molecule, partial_charge_method: str | None = None, use_conformers: list[Quantity] | None = None, strict_n_conformers: bool = False, normalize_partial_charges: bool = 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 for which partial charges are to be computed
- **partial_charge_method** – The charge model to use. One of [‘mmff94’, ‘gasteiger’]. 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** – 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** – 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** – 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** – 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: bool = False, hydrogens_are_explicit: bool = 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** – An RDKit molecule
- **allow_undefined_stereo** – If false, raises an exception if rdmol contains undefined stereochemistry.
- **hydrogens_are_explicit** – If False, RDKit will perform hydrogen addition using Chem.AddHs
- **_cls** – Molecule constructor

Returns

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:* Molecule, *aromaticity_model:* *str* = *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 – The aromaticity model to use. Only OEAroModel_MDL is supported.

Returns

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:* Molecule, *fixed_hydrogens:* *bool* = *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** – The molecule to convert into a SMILES.
- **fixed_hydrogens** – 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 – The InChI string of the molecule.

to_inchikey(*molecule:* Molecule, *fixed_hydrogens:* *bool* = *False*) → str

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** – The molecule to convert into a SMILES.
- **fixed_hydrogens** – 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 – The InChIKey representation of the molecule.

get_tagged_smarts_connectivity(*smarts:* *str*)

Returns a tuple of tuples indicating connectivity between tagged atoms in a SMARTS string. Does not return bond order.

Parameters
smarts – The tagged SMARTS to analyze

Returns

- *unique_tags* – A sorted tuple of all unique tagged atom map indices.
- *tagged_atom_connectivity* –

  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:* Molecule, *smarts:* *str*, *aromaticity_model:* *str* = *'OEAroModel_MDL'*, *unique:* *bool* = *False*) → list[tuple[int, ...]]

Find all SMARTS matches for the specified molecule, using the specified aromaticity model.

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

Parameters

- **molecule** – The molecule for which all specified SMARTS matches are to be located
- **smarts** – SMARTS string with optional SMIRKS-style atom tagging

- **aromaticity_model** – Molecule is prepared with this aromaticity model prior to querying.
- **unique** – If True, only return unique matches. If False, return all matches.
- : (.. 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 – The molecule containing the atom of interest

Returns
is_in_ring – 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 – The molecule containing the atom of interest

Returns
is_in_ring – 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 – 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 – The version of the wrapped toolkit

14.1.5 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.
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

SUPPORTED_CHARGE_METHODS	
supported_charge_methods	
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() → bool

Check whether the AmberTools toolkit is installed

Returns

is_installed – True if AmberTools is installed, False otherwise.

assign_partial_charges(*molecule: Molecule, partial_charge_method: str | None = None, use_conformers: list[Quantity] | None = None, strict_n_conformers: bool = False, normalize_partial_charges: bool = 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 for which partial charges are to be computed
- **partial_charge_method** – The charge model to use. One of ['gasteiger', 'am1bcc', 'am1-mulliken']. If None, 'am1-mulliken' will be used.
- **use_conformers** – 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** – 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** – 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** – 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** –

assign_fractional_bond_orders(*molecule: Molecule, bond_order_model: str | None = None, use_conformers: list[str] | None = 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** – The molecule to assign wiberg bond orders to

- **bond_order_model** – The charge model to use. Only allowed value is 'am1-wiberg'. If None, 'am1-wiberg' will be used.
- **use_conformers** – The conformers to use for fractional bond order calculation. If None, an appropriate number of conformers will be generated by an available ToolkitWrapper.
- **_cls** – 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** – The file to read the molecule from
- **file_format** – 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** – If false, raises an exception if any molecules contain undefined stereochemistry.
- **_cls** – Molecule constructor

Returns

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** – The file-like object to read the molecule from
- **file_format** – 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** – If false, raises an exception if any molecules contain undefined stereochemistry. If false, the function skips loading the molecule.
- **_cls** – Molecule constructor

Returns

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[str]

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 – 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 – The version of the wrapped toolkit

14.1.6 NAGLToolkitWrapper

class openff.toolkit.utils.toolkits.**NAGLToolkitWrapper**

NAGL toolkit wrapper for applying partial charges with a GCN model.

OpenFF NAGL computes partial charges directly from the molecular graph and independent of conformer coordinates using a Graph Convolutional Network.

__init__()

Methods

__init__()	
assign_partial_charges(molecule, ...[, ...])	Compute partial charges with NAGL and store in self.partial_charges
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

`supported_charge_methods`	
`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() → bool

Check whether the corresponding toolkit can be imported

Returns

is_installed – True if corresponding toolkit is installed, False otherwise.

assign_partial_charges(*molecule: Molecule, partial_charge_method: str, use_conformers: list['Quantity'] | None = None, strict_n_conformers: bool = False, normalize_partial_charges: bool = True, _cls: type['FrozenMolecule'] | None = None*)

Compute partial charges with NAGL and store in `self.partial_charges`

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

Parameters

- **molecule** – Molecule for which partial charges are to be computed
- **partial_charge_method** – The NAGL model to use. May be a path or the name of a model in a directory from the `openforcefield.nagl_model_path` entry point.
- **use_conformers** – This argument is ignored as NAGL does not generate or consider coordinates during inference.
- **strict_n_conformers** – This argument is ignored as NAGL does not generate or consider coordinates during inference.
- **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

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** – The file to read the molecule from
- **file_format** – 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** – If false, raises an exception if any molecules contain undefined stereochemistry.
- **_cls** – Molecule constructor

Returns

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** – The file-like object to read the molecule from
- **file_format** – 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** – If false, raises an exception if any molecules contain undefined stereochemistry. If false, the function skips loading the molecule.
- **_cls** – Molecule constructor

Returns

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[str]

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 – 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 – The version of the wrapped toolkit

14.1.7 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

PARTIAL_CHARGE_METHODS	
supported_charge_methods	
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: FrozenMolecule, partial_charge_method: str | None = None, use_conformers: Quantity | None = None, strict_n_conformers: bool = False, normalize_partial_charges: bool = True, _cls: type | None = 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 for which partial charges are to be computed
- **partial_charge_method** – The charge model to use. One of ['zeros', 'formal_charge']. If None, 'formal_charge' will be used.
- **use_conformers** – (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** – 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** – 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** – 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** – The file to read the molecule from
- **file_format** – 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** – If false, raises an exception if any molecules contain undefined stereochemistry.
- **_cls** – Molecule constructor

Returns

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** – The file-like object to read the molecule from
- **file_format** – 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** – If false, raises an exception if any molecules contain undefined stereochemistry. If false, the function skips loading the molecule.
- **_cls** – Molecule constructor

Returns

molecules – a list of Molecule objects is returned.

classmethod is_available()

Check whether the corresponding toolkit can be imported

Returns

is_installed – 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[str]

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 – 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 – The version of the wrapped toolkit

toolkit_registry_manager	A context manager that temporarily changes the ToolkitWrappers in the GLOBAL_TOOLKIT_REGISTRY.

14.1.8 toolkit_registry_manager

openff.toolkit.utils.toolkit_registry.**toolkit_registry_manager**(*toolkit_registry:* ToolkitRegistry | ToolkitWrapper)

A context manager that temporarily changes the ToolkitWrappers in the GLOBAL_TOOLKIT_REGISTRY. This can be useful in cases where one would otherwise need to otherwise manually specify the use of a specific ToolkitWrapper or ToolkitRegistry repeatedly in a block of code, or in cases where there isn't another way to switch the ToolkitWrappers used for a particular operation.

Examples

```
>>> from openff.toolkit import Molecule, RDKitToolkitWrapper, AmberToolsToolkitWrapper
>>> from openff.toolkit.utils import toolkit_registry_manager, ToolkitRegistry
>>> mol = Molecule.from_smiles("CCO")
>>> print(mol.to_smiles()) # This will use the OpenEye backend (if installed and␣
↪licensed)
[H]C([H])([H])C([H])([H])O[H]
>>> with toolkit_registry_manager(ToolkitRegistry([RDKitToolkitWrapper()])):
...     print(mol.to_smiles())
[H][O][C]([H])([H])[C]([H])([H])[H]
```

14.2 Serialization support

`Serializable`	Mix-in to add serialization and deserialization support via JSON, YAML, BSON, TOML, MessagePack, and XML.

14.2.1 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 – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – 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 – A JSON serialized representation of the object

Returns
instance – An instantiated object

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns
serialized – 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 – A BSON serialized representation of the object

Returns
instance – An instantiated object

to_toml()

Return a TOML serialized representation.

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

Returns
serialized – 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 – A TOML serialized representation of the object

Returns
instance – An instantiated object

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns
serialized – A YAML serialized representation of the object

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters
serialized – A YAML serialized representation of the object

Returns
instance – Instantiated object

to_messagepack()

Return a MessagePack representation.

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

Returns
serialized – 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 – A MessagePack-encoded bytes serialized representation

Returns
instance – Instantiated object.

to_xml(*indent=2*)

Return an XML representation.

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

Parameters
indent – If not None, will pretty-print with specified number of spaces for indentation

Returns
serialized – 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 – An XML serialized representation

Returns
instance – 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 – 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 – A pickled representation of the object

Returns
instance – An instantiated object

14.3 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.

14.3.1 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** – A sequence of elements.
- **converter** – Functions that will be used to convert each new element of the list.

- **validator** – 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.

14.3.2 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** – A mapping of elements, probably a dict.
- **converter** – Functions that will be used to convert each new element of the dict.

- **validator** – 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, default 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, default 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

14.4 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	

14.4.1 inherit_docstrings

openff.toolkit.utils.utils.**inherit_docstrings**(*cls*)

Inherit docstrings from parent class

14.4.2 all_subclasses

openff.toolkit.utils.utils.**all_subclasses**(*cls: type[T]*) → list[type[T]]

Recursively retrieve all subclasses of the specified class

14.4.3 temporary_cd

openff.toolkit.utils.utils.**temporary_cd**(*dir_path*)

Context to temporary change the working directory.

Parameters

dir_path – The directory path to enter within the context

Examples

```
>>> dir_path = '/tmp'
>>> with temporary_cd(dir_path):
...     pass  # do something in dir_path
```

14.4.4 get_data_file_path

openff.toolkit.utils.utils.**get_data_file_path**(*relative_path: str*) → str

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

relative_path – Name of the file to load (with respect to *openff/toolkit/data/*)

14.4.5 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 – 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

14.4.6 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 – 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

14.4.7 get_molecule_parameterIDs

openff.toolkit.utils.utils.**get_molecule_parameterIDs**(*molecules: list['Molecule'], forcefield:* ForceField) → tuple[dict, dict]

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 molecules (with explicit hydrogens) to parse
- **forcefield** – The ForceField to apply

Returns

- *parameters_by_molecule* – 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* – 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.

14.4.8 unit_to_string

openff.toolkit.utils.utils.**unit_to_string**(*input_unit: Unit*) → str

14.5 Exceptions

Exceptions raised by the Toolkit.

exceptions

14.5.1 openff.toolkit.utils.exceptions

Exceptions

AmbiguousAtomChemicalAssignment(res_name, ...)	Exception raised when substructure does not contain enough information
AmbiguousBondChemicalAssignment(res_name, ...)	Exception raised when substructure does not contain enough information
AntechamberNotFoundError(msg)	The antechamber executable was not found

continues on next page

Table 1 – continued from previous page

AtomMappingWarning	A warning when dealing with atom maping or indices.
AtomNotInTopologyError(msg)	An atom was not found in a topology.
BondExistsError(msg)	The program attempted to add a bond that already exists
BondNotInTopologyError(msg)	An bond was not found in a topology.
CallbackRegistrationError(msg)	Error raised when callback registration fails.
ChargeCalculationError(msg)	An unhandled error occurred in an external toolkit during charge calculation
ChargeMethodUnavailableError(msg)	A toolkit does not support the requested partial_charge_method combination
ChemicalEnvironmentParsingError(msg)	Exception for when SMARTS/SMIRKS are not parseable by a wrapped toolkit
ConformerGenerationError(msg)	Conformer generation via a wrapped toolkit failed.
ConstraintExsistsError(msg)	Attempting to override a constraint that already exists with a specified distance.
DuplicateParameterError(msg)	Exception raised when trying to add a ParameterType that already exists
DuplicateUniqueMoleculeError(msg)	Exception for when the user provides indistinguishable unique molecules when trying to identify atoms from a PDB
DuplicateVirtualSiteTypeException(msg)	Exception raised when trying to register two different virtual site classes with the same 'type'
FractionalBondOrderInterpolationMethodUnsuppor	Exception for when an unsupported fractional bond order interpolation assignment method is called.
GAFFAtomTypeWarning	A warning raised if a loaded mol2 file possibly uses GAFF atom types.
HierarchyIteratorNameConflictError(msg)	Exception raised when trying to access a hierarchy scheme with a name that already exists as a *Topology* or *Molecule* attribute.
HierarchySchemeNotFoundException(msg)	Exception raised when trying to access a HierarchyScheme from a molecule that doesn't have one with the given iterator name
HierarchySchemeWithIteratorNameAlreadyRegister	Exception raised when trying to add a HierarchyScheme to a molecule that already has one with the same iterator name
InChIParseError(msg)	The InChI record could not be parsed.
IncompatibleParameterError(msg)	Exception for when a set of parameters is scientifically/technically incompatible with another
IncompatibleShapeError(msg)	Exception for when a value is in the wrong shape
IncompatibleTypeError(msg)	Exception for when a value is in an incompatible type
IncompatibleUnitError(msg)	Exception for when a parameter is in the wrong units for a ParameterHandler's unit system
InconsistentStereochemistryError(msg)	Error raised when stereochemistry is inconsistent before and after conversions between molecule representations.
IncorrectNumConformersError(msg)	The requested partial_charge_method expects a different number of conformers than was provided
IncorrectNumConformersWarning	The requested partial_charge_method expects a different number of conformers than was provided

continues on next page

Table 1 – continued from previous page

InvalidAromaticityModelError(msg)	General exception for errors while setting the aromaticity model of a Topology.
InvalidAtomMetadataError(msg)	The program attempted to set atom metadata to an invalid type
InvalidBondOrderError(msg)	Exception for passing a non-int to *Molecule.bond_order*
InvalidBoxVectorsError(msg)	Exception for setting invalid box vectors
InvalidConformerError(msg)	This error is raised when the conformer added to the molecule has a different connectivity to that already defined.
InvalidIUPACNameError(msg)	Failed to parse IUPAC name
InvalidPeriodicityError(msg)	Exception for setting invalid periodicity
InvalidQCInputError(msg)	This error is raised when an input to Molecule.from_qcschema is invalid.
InvalidToolkitError(msg)	A non-toolkit object was received when a toolkit object was expected
InvalidToolkitRegistryError(msg)	An object other than a ToolkitRegistry or toolkit wrapper was received
LicenseError(msg)	This function requires a license that cannot be found.
MissingCMILESError(msg)	Error raised when attempting to convert an QC input to an OFF Molecule, but the CMILES can't be found or isn't present.
MissingConformersError(msg)	Error raised when a molecule is missing conformer(s) in a context in which it is expected to have them.
MissingIndexedAttributeError(msg)	Error raised when an indexed attribute does not exist
MissingPackageError(msg)	This function requires a package that is not installed.
MissingPartialChargesError(msg)	Error raised when a molecule is missing partial charges in a context in which it is expected to have them.
MissingUniqueMoleculesError(msg)	Exception for when unique_molecules is required but not found
MoleculeNotInTopologyError(msg)	A molecule was not found in a topology.
MoleculeParseError(msg)	The molecule could not be created from the given format
MultipleMoleculesInPDBError(msg)	Error raised when a multiple molecules are found when one was expected
NonUniqueSubstructureName(duplicate_keys)	Exception raised when nonunique names are given
NotAttachedToMoleculeError(msg)	Exception for when a component does not belong to a Molecule object, but is queried
NotBondedError(msg)	Exception for when a function requires a bond between two atoms, but none is present
NotEnoughPointsForInterpolationError(msg)	Exception for when less than two points are provided for interpolation
NotInTopologyError(msg)	An object was not found in a topology.
OpenEyeError(msg)	Error raised when an OpenEye Toolkits operation fails.
OpenEyeImportError(msg)	Error raised when importing an OpenEye module fails.

continues on next page

Table 1 – continued from previous page

OpenFFToolkitException(msg)	Base exception for custom exceptions raised by the OpenFF Toolkit
ParameterHandlerRegistrationError(msg)	Exception for errors in ParameterHandler registration
ParameterLookupError(msg)	Exception raised when something goes wrong in a parameter lookup in ParameterHandler.__getitem__
PartialChargeVirtualSitesError(msg)	Exception thrown when partial charges cannot be computed for a Molecule because the ForceField applies virtual sites.
RadicalsNotSupportedError(msg)	The OpenFF Toolkit does not currently support parsing molecules with radicals.
RemapIndexError(msg)	An error with indices used to remap a molecule
SMILESParseError(msg)	The record could not be parsed into the given format
SMIRKSMismatchError(msg)	Exception for cases where smirks are inappropriate for the environment type they are being parsed into
SMIRKSParsingError(msg)	Exception for when SMIRKS are not parseable for any environment
SMIRNOFFAromaticityError(msg)	Exception thrown when an incompatible SMIRNOFF aromaticity model is checked for compatibility.
SMIRNOFFParseError(msg)	Error for when a SMIRNOFF data structure is not parseable by a ForceField
SMIRNOFFSpecError(msg)	Exception for when data is noncompliant with the SMIRNOFF data specification.
SMIRNOFFSpecUnimplementedError(msg)	Exception for when a portion of the SMIRNOFF specification is not yet implemented.
SMIRNOFFVersionError(msg)	Exception thrown when an incompatible SMIRNOFF version data structure is attempted to be read.
SmilesParsingError(msg)	This error is raised when parsing a SMILES string results in an error.
SubstructureAtomSmartsInvalid(name, ...)	Exception raised when atom or bond smarts are found to be improperly formatted
SubstructureBondSmartsInvalid(name, bond, ...)	
SubstructureImproperlySpecified(name, reason)	Exception raised when substructure does not contain enough information
ToolkitUnavailableException(msg)	The requested toolkit is unavailable.
UnassignedAngleParameterException(msg)	Exception raised when there are angle terms for which a ParameterHandler can't find parameters.
UnassignedBondParameterException(msg)	Exception raised when there are bond terms for which a ParameterHandler can't find parameters.
UnassignedChemistryInPDBError([msg, ...])	Error raised when a bond or atom in a polymer could not be assigned chemical information.
UnassignedMoleculeChargeException(msg)	Exception raised when no charge method is able to assign charges to a molecule.
UnassignedProperTorsionParameterException(msg)	Exception raised when there are proper torsion terms for which a ParameterHandler can't find parameters.

continues on next page

Table 1 – continued from previous page

UnassignedValenceParameterException(msg)	Exception raised when there are valence terms for which a ParameterHandler can't find parameters.
UndefinedStereochemistryError(msg)	A molecule was attempted to be loaded with undefined stereochemistry
UnsupportedFileTypeError(msg)	Error raised when attempting to parse an unsupported file type.
UnsupportedMoleculeConversionError(msg)	Error raised when attempting to instantiate a Molecule with insufficient inputs.
VirtualSitesUnsupportedError(msg)	Exception raised when trying to store virtual sites in a *Molecule* or *Topology* object.
WrongShapeError(msg)	Error raised when an array of the wrong shape is found

exception openff.toolkit.utils.exceptions.**OpenFFToolkitException**(*msg: str*)

Base exception for custom exceptions raised by the OpenFF Toolkit

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**IncompatibleUnitError**(*msg: str*)

Exception for when a parameter is in the wrong units for a ParameterHandler's unit system

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**IncompatibleShapeError**(*msg: str*)

Exception for when a value is in the wrong shape

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**IncompatibleTypeError**(*msg: str*)

Exception for when a value is in an incompatible type

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**MissingPackageError**(*msg: str*)

This function requires a package that is not installed.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**ToolkitUnavailableException**(*msg: str*)

The requested toolkit is unavailable.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**LicenseError**(*msg: str*)

This function requires a license that cannot be found.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**InvalidToolkitError**(*msg: str*)

A non-toolkit object was received when a toolkit object was expected

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**InvalidToolkitRegistryError**(*msg: str*)

An object other than a ToolkitRegistry or toolkit wrapper was received

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**UndefinedStereochemistryError**(*msg: str*)

A molecule was attempted to be loaded with undefined stereochemistry

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**GAFFAtomTypeWarning**

A warning raised if a loaded mol2 file possibly uses GAFF atom types.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**ChargeMethodUnavailableError**(*msg: str*)

A toolkit does not support the requested partial_charge_method combination

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**IncorrectNumConformersError**(*msg: str*)

The requested partial_charge_method expects a different number of conformers than was provided

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**IncorrectNumConformersWarning**

The requested partial_charge_method expects a different number of conformers than was provided

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**ChargeCalculationError**(*msg: str*)

An unhandled error occurred in an external toolkit during charge calculation

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**ConformerGenerationError**(*msg: str*)

Conformer generation via a wrapped toolkit failed.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**InvalidIUPACNameError**(*msg: str*)

Failed to parse IUPAC name

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**AntechamberNotFoundError**(*msg: str*)

The antechamber executable was not found

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**MoleculeParseError**(*msg: str*)

The molecule could not be created from the given format

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**SMILESParseError**(*msg: str*)

The record could not be parsed into the given format

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**AtomMappingWarning**

A warning when dealing with atom maping or indices.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**InChIParseError**(*msg: str*)

The InChI record could not be parsed.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**RadicalsNotSupportedError**(*msg: str*)

The OpenFF Toolkit does not currently support parsing molecules with radicals.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**InvalidConformerError**(*msg: str*)

This error is raised when the conformer added to the molecule has a different connectivity to that already defined. or any other conformer related issues.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**InvalidQCInputError**(*msg: str*)

This error is raised when an input to Molecule.from_qcschema is invalid.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**SmilesParsingError**(*msg: str*)

This error is raised when parsing a SMILES string results in an error.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**NotAttachedToMoleculeError**(*msg: str*)

Exception for when a component does not belong to a Molecule object, but is queried

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**NotInTopologyError**(*msg: str*)

An object was not found in a topology.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**RemapIndexError**(*msg: str*)

An error with indices used to remap a molecule

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**AtomNotInTopologyError**(*msg: str*)

An atom was not found in a topology.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**BondNotInTopologyError**(*msg: str*)

An bond was not found in a topology.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**MoleculeNotInTopologyError**(*msg: str*)

A molecule was not found in a topology.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**InvalidAtomMetadataError**(*msg: str*)

The program attempted to set atom metadata to an invalid type

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**BondExistsError**(*msg: str*)

The program attempted to add a bond that already exists

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**ConstraintExsistsError**(*msg: str*)

Attempting to override a constraint that already exists with a specified distance.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**DuplicateUniqueMoleculeError**(*msg: str*)

Exception for when the user provides indistinguishable unique molecules when trying to identify atoms from a PDB

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**NotBondedError**(*msg: str*)

Exception for when a function requires a bond between two atoms, but none is present

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**InvalidBondOrderError**(*msg: str*)

Exception for passing a non-int to *Molecule.bond_order*

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**InvalidBoxVectorsError**(*msg: str*)

Exception for setting invalid box vectors

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**InvalidPeriodicityError**(*msg: str*)

Exception for setting invalid periodicity

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**MissingUniqueMoleculesError**(*msg: str*)

Exception for when unique_molecules is required but not found

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**SMIRKSMismatchError**(*msg: str*)

Exception for cases where smirks are inappropriate for the environment type they are being parsed into

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**SMIRKSParsingError**(*msg: str*)

Exception for when SMIRKS are not parseable for any environment

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**ChemicalEnvironmentParsingError**(*msg: str*)

Exception for when SMARTS/SMIRKS are not parseable by a wrapped toolkit

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**ParameterHandlerRegistrationError**(*msg: str*)

Exception for errors in ParameterHandler registration

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**SMIRNOFFVersionError**(*msg: str*)

Exception thrown when an incompatible SMIRNOFF version data structure is attempted to be read.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**SMIRNOFFAromaticityError**(*msg: str*)

Exception thrown when an incompatible SMIRNOFF aromaticity model is checked for compatibility.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**InvalidAromaticityModelError**(*msg: str*)

General exception for errors while setting the aromaticity model of a Topology.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**SMIRNOFFParseError**(*msg: str*)

Error for when a SMIRNOFF data structure is not parseable by a ForceField

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**PartialChargeVirtualSitesError**(*msg: str*)

Exception thrown when partial charges cannot be computed for a Molecule because the ForceField applies virtual sites.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**SMIRNOFFSpecError**(*msg: str*)

Exception for when data is noncompliant with the SMIRNOFF data specification.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**SMIRNOFFSpecUnimplementedError**(*msg: str*)

Exception for when a portion of the SMIRNOFF specification is not yet implemented.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**FractionalBondOrderInterpolationMethodUnsupportedError**(*msg: str*)

Exception for when an unsupported fractional bond order interpolation assignment method is called.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**NotEnoughPointsForInterpolationError**(*msg: str*)

Exception for when less than two points are provided for interpolation

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**IncompatibleParameterError**(*msg: str*)

Exception for when a set of parameters is scientifically/technically incompatible with another

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**UnassignedValenceParameterException**(*msg: str*)

Exception raised when there are valence terms for which a ParameterHandler can't find parameters.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**UnassignedBondParameterException**(*msg: str*)

Exception raised when there are bond terms for which a ParameterHandler can't find parameters.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**UnassignedAngleParameterException**(*msg: str*)

Exception raised when there are angle terms for which a ParameterHandler can't find parameters.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**UnassignedProperTorsionParameterException**(*msg: str*)

Exception raised when there are proper torsion terms for which a ParameterHandler can't find parameters.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**UnassignedMoleculeChargeException**(*msg: str*)

Exception raised when no charge method is able to assign charges to a molecule.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**DuplicateParameterError**(*msg: str*)

Exception raised when trying to add a ParameterType that already exists

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**ParameterLookupError**(*msg: str*)

Exception raised when something goes wrong in a parameter lookup in ParameterHandler.__getitem__

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**DuplicateVirtualSiteTypeException**(*msg: str*)

Exception raised when trying to register two different virtual site classes with the same 'type'

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**CallbackRegistrationError**(*msg: str*)

Error raised when callback registration fails.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**HierarchySchemeWithIteratorNameAlreadyRegisteredException**(*msg: str*)

Exception raised when trying to add a HierarchyScheme to a molecule that already has one with the same iterator name

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**HierarchySchemeNotFoundException**(*msg: str*)

Exception raised when trying to access a HierarchyScheme from a molecule that doesn't have one with the given iterator name

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**HierarchyIteratorNameConflictError**(*msg: str*)

Exception raised when trying to access a hierarchy scheme with a name that already exists as a *Topology* or *Molecule* attribute.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**VirtualSitesUnsupportedError**(*msg: str*)

Exception raised when trying to store virtual sites in a *Molecule* or *Topology* object.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**MissingIndexedAttributeError**(*msg: str*)

Error raised when an indexed attribute does not exist

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**MissingPartialChargesError**(*msg: str*)

Error raised when a molecule is missing partial charges in a context in which it is expected to have them.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**MissingConformersError**(*msg: str*)

Error raised when a molecule is missing conformer(s) in a context in which it is expected to have them.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**MissingCMILESError**(*msg: str*)

Error raised when attempting to convert an QC input to an OFF Molecule, but the CMILES can't be found or isn't present.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**UnsupportedMoleculeConversionError**(*msg: str*)

Error raised when attempting to instantiate a Molecule with insufficient inputs.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**InconsistentStereochemistryError**(*msg: str*)

Error raised when stereochemistry is inconsistent before and after conversions between molecule representations.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**UnsupportedFileTypeError**(*msg: str*)

Error raised when attempting to parse an unsupported file type.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**OpenEyeError**(*msg: str*)

Error raised when an OpenEye Toolkits operation fails.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**OpenEyeImportError**(*msg: str*)

Error raised when importing an OpenEye module fails.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**MultipleMoleculesInPDBError**(*msg: str*)

Error raised when a multiple molecules are found when one was expected

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**WrongShapeError**(*msg: str*)

Error raised when an array of the wrong shape is found

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**UnassignedChemistryInPDBError**(*msg: str | None = None, substructure_library: dict[str, list[tuple]] | None = None, omm_top: OpenMMTopology | None = None, unassigned_bonds: list[tuple[int, int]] | None = None, unassigned_atoms: list[int] | None = None, matches: DefaultDict[int, list[str]] | None = None*)

Error raised when a bond or atom in a polymer could not be assigned chemical information.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**NonUniqueSubstructureName**(*duplicate_keys*)

Exception raised when nonunique names are given

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**SubstructureAtomSmartsInvalid**(*name*, *atom_smarts*, *smarts*, *reason*)

Exception raised when atom or bond smarts are found to be improperly formatted

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**SubstructureBondSmartsInvalid**(*name*, *bond*, *valid_bond_types*)

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**SubstructureImproperlySpecified**(*name*, *reason*)

Exception raised when substructure does not contain enough information

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**AmbiguousAtomChemicalAssignment**(*res_name*, *mol_atom*, *query_atom*, *reason*)

Exception raised when substructure does not contain enough information

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openff.toolkit.utils.exceptions.**AmbiguousBondChemicalAssignment**(*res_name*, *mol_bond*, *query_bond*, *reason*)

Exception raised when substructure does not contain enough information

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

PYTHON MODULE INDEX

O

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

P

R

S

T

U

V

W

X