

---

# **openforcefield Documentation**

*Release 0.5.1*

**Open Force Field Consortium**

**Sep 24, 2019**



## CONTENTS

<b>1</b>	<b>User Guide</b>	<b>3</b>
<b>2</b>	<b>API documentation</b>	<b>39</b>
	<b>Index</b>	<b>413</b>



A modern, extensible library for molecular mechanics force field science from the [Open Force Field Initiative](#)



## 1.1 Installation

### 1.1.1 Installing via *conda*

The simplest way to install the Open Forcefield Toolkit is via the [conda](#) package manager. Packages are provided on the [omnia Anaconda Cloud channel](#) for Linux, OS X, and Win platforms. The [openforcefield Anaconda Cloud page](#) has useful instructions and [download statistics](#).

If you are using the [anaconda](#) scientific Python distribution, you already have the conda package manager installed. If not, the quickest way to get started is to install the [miniconda](#) distribution, a lightweight minimal installation of Anaconda Python.

On linux, you can install the Python 3 version into `$HOME/miniconda3` with (on bash systems):

```
$ wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ bash ./Miniconda3-latest-Linux-x86_64.sh -b -p $HOME/miniconda3
$ source ~/miniconda3/etc/profile.d/conda.sh
$ conda activate base
```

On osx, you want to use the osx binary

```
$ curl https://repo.continuum.io/miniconda/Miniconda3-latest-MacOSX-x86_64.sh -O
$ bash ./Miniconda3-latest-MacOSX-x86_64.sh -b -p $HOME/miniconda3
$ source ~/miniconda3/etc/profile.d/conda.sh
$ conda activate base
```

You may want to add the new `source ~/miniconda3/etc/profile.d/conda.sh` line to your `~/.bashrc` file to ensure Anaconda Python can be enabled in subsequent terminal sessions. `conda activate base` will need to be run in each subsequent terminal session to return to the environment where the toolkit will be installed.

Note that openforcefield will be installed into this local Python installation, so that you will not need to worry about disrupting existing Python installations.

---

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

---

### 1.1.2 Required dependencies

The openforcefield toolkit makes use of the [Omnia](#) and [Conda Forge](#) free and open source community package repositories:

```
$ conda config --add channels omnia --add channels conda-forge
$ conda update --all
```

This only needs to be done once.

---

**Note:** If automation is required, provide the `--yes` argument to `conda update` and `conda install` commands. More information on the conda command-line API can be found in the [conda online documentation](#).

---

### Release build

You can install the latest stable release build of openforcefield via the conda package with

```
$ conda config --add channels omnia --add channels conda-forge
$ conda install openforcefield
```

This version is recommended for all users not actively developing new forcefield parameterization algorithms.

---

**Note:** The conda package manager will install dependencies from binary packages automatically, including difficult-to-install packages such as OpenMM, numpy, and scipy. This is really the easiest way to get started.

---

### Upgrading your installation

To update an earlier conda installation of openforcefield to the latest release version, you can use conda update:

```
$ conda update openforcefield
```

### Optional dependencies

This 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 (provided you have a valid license file):

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

No essential openforcefield release capabilities *require* the OpenEye toolkit, but the Open Force Field developers make use of it in parameterizing new open source force fields. It is known that there are certain differences in toolkit behavior between RDKit 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](#).

### 1.1.3 Alternative method: Single-file installer

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

The installers are between 200 and 300 MB each, and can be downloaded from the “Assets” section of the Toolkit’s [GitHub Releases page](#). They are generated using a [workflow leveraging the “conda constructor” utility](#).

Please report any installer difficulties to the [OFF Toolkit Issue tracker](#), as we hope to make this a major distribution channel for the toolkit moving forward.

## Installation

Download the appropriate installer (openforcefield-<X.Y.Z>-py37-<your platform>-x86\_64.sh) from the “Assets” section at the bottom of the desired release. Then, install the toolkit with the following command:

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

and follow the prompts.

**Note:** You must have write access to the installation directory. This is generally somewhere in the user’s home directory. When prompted, we recommend NOT making modifications to your bash\_profile.

**Warning:** We recommend that you do not install this package as root. Conda is intended to support on-the-fly creation of several independent environments, and [managing a multi-user conda installation is somewhat involved](#).

## Usage

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

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

Once the base environment is activated, your system will default to use python (and other executables) from the newly installed conda environment.

## Installing optional OpenEye toolkits

We're waiting on permission to redistribute the OpenEye toolkits inside the single-file installer, so for now the installers only ship with the open-source backend (RDKit+AmberTools). With this in mind, the conda environment created by the installer *contains the conda package manager itself*, which can be used to install the OpenEye toolkits if you have access to the Anaconda cloud.

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

---

**Note:** The OpenEye Toolkits conda package still requires a valid OpenEye license file to run.

---

## 1.2 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

### 1.2.1 0.5.1 - Adding the parameter coverage example notebook

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

The parameter coverage example notebook goes hand-in-hand with the release candidate of our initial force field, [openff-1.0.0-RC1.offxml](#), which will be temporarily available until the official force field release is made in October. Our goal in publishing this notebook alongside our first major refitting is to allow interested users to check whether there is parameter coverage for their molecules of interest. If the force field is unable to parameterize a molecule, this notebook will generate reports of the specific chemistry that is not covered. We understand that many organizations in our field have restrictions about sharing specific molecules, and the outputs from this notebook can easily be cropped to communicate unparameterizable chemistry without revealing the full structure.

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

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

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

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

Complete details about this release are below.

### Example added

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

### New features

- [PR #419](#): Unassigned valence parameter exceptions now include a list of tuples of [TopologyAtom](#) which were unable to be parameterized (`exception.unassigned_topology_atom_tuples`) and the class of the [ParameterHandler](#) that raised the exception (`exception.handler_class`).
- [PR #425](#): Implements Trevor Gokey's suggestion from [Issue #411](#), which enables pickling of [ForceFields](#) and [ParameterHandlers](#). Note that, while XML representations of "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.

### Improved documentation and warnings

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

### Bugfixes

- [PR #419](#): Fixes [Issue #417](#) and [Issue #418](#), where [RDKitToolkitWrapper.from\\_file](#) would disregard the `allow_undefined_stereo` kwarg and skip the first molecule when reading a SMILES file.

### Files removed

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

### 1.2.2 0.5.0 - GBSA support and quality-of-life improvements

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

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

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

#### New features

- [PR #363](#): Implements [GBSAHandler](#), which supports the [GBSA tag in the SMIRNOFF specification](#). Currently, only [GBSAHandlers](#) with `gb_model="OBC2"` support setting non-default values for the `surface_area_penalty` term (default  $5.4 \times \text{calories/mole/angstroms}^2$ ), 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  $1.4 \times \text{angstroms}$ . Files containing experimental SMIRNOFF-format implementations of HCT, OBC1, and OBC2 are included with this release (see below). Additional details of these models, including literature references, are available on the [SMIRNOFF specification page](#).

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

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

#### Bugfixes

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

## Files added

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

## 1.2.3 0.4.1 - Bugfix Release

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

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

## Bug fixes

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

---

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

---

## Code enhancements

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

## Force fields added

- [PR #368](#): Temporarily adds `test_forcefields/smirnoff99frosst_experimental.offxml` to address hierarchy problems, redundancies, SMIRKS pattern typos etc., as documented in [issue #367](#). Will ultimately be propagated to an updated forcefield 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.

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

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

## SMIRNOFF 0.3 specification updates

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

## Performance improvements and bugfixes

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

## New features

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

## API-breaking changes

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



## 1.2.5 0.3.0 - API Improvements

Several improvements and changes to public API.

### New features

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

### API-breaking Changes

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

### Bugfixes

- [PR #327](#): Fix units in `tip3p.offxml` (note that this file is still not loadable by current toolkit)
- [PR #325](#): Fix solvent box for provided test system to resolve periodic clashes.
- [PR #325](#): Add informative message containing Hill formula when a molecule can't be matched in `Topology.from_openmm`.
- [PR #325](#): Provide warning or error message as appropriate when a molecule is missing stereochemistry.
- [PR #316](#): Fix formatting issues in GBSA section of SMIRNOFF spec
- [PR #308](#): Cache molecule SMILES to improve system creation speed
- [PR #306](#): Allow single-atom molecules with all zero coordinates to be converted to OE/RDK mols
- [PR #313](#): Fix issue where constraints are applied twice to constrained bonds

## 1.2.6 0.2.2 - Bugfix release

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

### Bugfixes

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

## 1.2.7 0.2.1 - Bugfix release

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



## Bugfixes

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

## 1.2.8 0.2.0 - Initial RDKit support

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

### New features

- Major overhaul, resulting in the creation of the [SMIRNOFF 0.2 specification](#) and its XML representation
- Updated API and infrastructure for reference SMIRNOFF ForceField implementation
- Implementation of modular ParameterHandler classes which process the topology to add all necessary forces to the system.
- Implementation of modular ParameterIOHandler classes for reading/writing different serialized SMIRNOFF forcefield 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

## 1.2.9 0.1.0

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

### New features

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

## Bugfixes

### Treatment of improper torsions

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

## 1.3 The SMIRKs Native Open Force Field (SMIRNOFF) specification

SMIRNOFF is a specification for encoding molecular mechanics force fields from the [Open Force Field Initiative](#) based on direct chemical perception using the broadly-supported [SMARTS](#) language, utilizing atom tagging extensions from [SMIRKS](#).

### 1.3.1 Authors and acknowledgments

The SMIRNOFF specification was designed by the [Open Force Field Initiative](#).

Primary contributors include:

- Caitlin C. Bannan (University of California, Irvine) <bannanc@uci.edu>
- Christopher I. Bayly (OpenEye Software) <bayly@eyesopen.com>
- John D. Chodera (Memorial Sloan Kettering Cancer Center) <john.chodera@choderalab.org>
- David L. Mobley (University of California, Irvine) <dmobley@uci.edu>

SMIRNOFF and its reference implementation in the openforcefield toolkit was heavily inspired by the [ForceField](#) class from the [OpenMM](#) molecular simulation package, and its associated [XML format](#), developed by [Peter K. Eastman](#) (Stanford University).

### 1.3.2 Representations and encodings

A force field in the SMIRNOFF format can be encoded in multiple representations. Currently, only an [XML](#) representation is supported by the reference implementation of the [openforcefield toolkit](#).

#### XML representation

A SMIRNOFF force field can be described in an [XML](#) representation, which provides a human- and machine-readable form for encoding the parameter set and its typing rules. This document focuses on describing the XML representation of the force field.

- By convention, XML-encoded SMIRNOFF force fields use an `.offxml` extension if written to a file to prevent confusion with other file formats.
- In XML, numeric quantities appear as strings, like "1" or "2.3".
- Integers should always be written without a decimal point, such as "1", "9".
- Non-integral numbers, such as parameter values, should be written with a decimal point, such as "1.23", "2.".
- In XML, certain special characters that occur in valid SMARTS/SMIRKS patterns (such as ampersand symbols &) must be specially encoded. See [this list of XML and HTML character entity references](#) for more details.

#### Future representations: JSON, MessagePack, YAML, and TOML

We are considering supporting [JSON](#), [MessagePack](#), [YAML](#), and [TOML](#) representations as well.

### 1.3.3 Reference implementation

A reference implementation of the SMIRNOFF XML specification is provided in the [openforcefield toolkit](#).

### 1.3.4 Support for molecular simulation packages

The reference implementation currently generates parameterized molecular mechanics systems for the GPU-accelerated [OpenMM](#) molecular simulation toolkit. Parameterized systems can subsequently be converted for use in other popular molecular dynamics simulation packages (including [AMBER](#), [CHARMM](#), [NAMD](#), [Desmond](#), and [LAMMPS](#)) via [ParmEd](#) and [InterMol](#). See [Converting SMIRNOFF parameterized systems to other simulation packages](#) for more details.

### 1.3.5 Basic structure

A reference implementation of a SMIRNOFF force field parser that can process XML representations (denoted by `.offxml` file extensions) can be found in the `ForceField` class of the `openforcefield.typing.engines.smirnoff` module.

Below, we describe the main structure of such an XML representation.

#### The enclosing <SMIRNOFF> tag

A SMIRNOFF forcefield XML specification always is enclosed in a `<SMIRNOFF>` tag, with certain required attributes provided. The required and permitted attributes defined in the `<SMIRNOFF>` are recorded in the `version` attribute, which describes the top-level attributes that are expected or permitted to be defined.

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

#### Versioning

The SMIRNOFF force field format supports versioning via the `version` attribute to the root `<SMIRNOFF>` tag, e.g.:

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

The version format is `x.y`, where `x` denotes the major version and `y` denotes the minor version. SMIRNOFF versions are guaranteed to be backward-compatible within the *same major version number series*, but it is possible major version increments will break backwards-compatibility.

SMIRNOFF tag version	Required attributes	Optional attributes
0.1	<code>aromaticity_model</code>	Date, Author
0.2	<code>aromaticity_model</code>	Date, Author
0.3	<code>aromaticity_model</code>	Date, Author

The SMIRNOFF tag versions describe the required and allowed force field-wide settings. The list of keywords is as follows:

## Aromaticity model

The `aromaticity_model` specifies the aromaticity model used for chemical perception (here, `OEArModel_MDL`).

Currently, the only supported model is `OEArModel_MDL`, which is implemented in both the RDKit and the OpenEye Toolkit.

---

**Note:** Add link to complete open specification of `OEArModel_MDL` aromaticity model.

---

## Metadata

Typically, date and author information is included:

```
<Date>2016-05-25</Date>
<Author>J. D. Chodera (MSKCC) charge increment tests</Author>
```

The `<Date>` tag should conform to [ISO 8601 date formatting guidelines](#), such as `2018-07-14` or `2018-07-14T08:50:48+00:00` (UTC time).

## Parameter generators

Within the `<SMIRNOFF>` tag, top-level tags encode parameters for a force field based on a SMARTS/SMIRKS-based specification describing the chemical environment the parameters are to be applied to. The file has tags corresponding to OpenMM force terms (Bonds, Angles, ProperTorsions, etc., as discussed in more detail below); these specify functional form and other information for individual force terms.

```
<Angles version="0.3" potential="harmonic">
...
</Angles>
```

which introduces the following Angle child elements which will use a harmonic potential.

## Specifying parameters

Under each of these force terms, there are tags for individual parameter lines such as these:

```
<Angles version="0.3" potential="harmonic">
  <Angle smirks="[a,A:1]-[#6X4:2]-[a,A:3]" angle="109.50*degree" k="100.0*kilocalorie_per_mole/
↪radian**2"/>
  <Angle smirks="[#1:1]-[#6X4:2]-[#1:3]" angle="109.50*degree" k="70.0*kilocalorie_per_mole/radian**2"/
↪>
</Angles>
```

The first of these specifies the `smirks` attribute as `[a,A:1]-[#6X4:2]-[a,A:3]`, specifying a SMIRKS pattern that matches three connected atoms specifying an angle. This particular SMIRKS pattern matches a tetravalent carbon at the center with single bonds to two atoms of any type. This pattern is essentially a [SMARTS](#) string with numerical atom tags commonly used in [SMIRKS](#) to identify atoms in chemically unique environments—these can be thought of as tagged regular expressions for identifying chemical environments, and atoms within those environments. Here, `[a,A]` denotes any atom—either aromatic (a) or aliphatic (A), while `[#6X4]` denotes a carbon by element number (#6) that with four substituents (X4). The symbol `-` joining these groups denotes a single bond. The strings `:1`, `:2`, and `:3` label these atoms as indices 1, 2, and 3, with

2 being the central atom. Equilibrium angles are provided as the angle attribute, along with force constants as the k attribute (with corresponding units included in the expression).

**Note:** The reference implementation of the SMIRNOFF specification implemented in the Open Force Field toolkit will, by default, raise an exception if an unexpected attribute is encountered. The toolkit can be configured to accept non-spec keywords, but these are considered “cosmetic” and will not be evaluated. For example, providing an `<Angle>` tag that also specifies a second force constant `k2` will result in an exception, unless the user specifies that “cosmetic” attributes should be accepted by the parser.

### SMIRNOFF parameter specification is hierarchical

Parameters that appear later in a SMIRNOFF specification override those which come earlier if they match the same pattern. This can be seen in the example above, where the first line provides a generic angle parameter for any tetravalent carbon (single bond) angle, and the second line overrides this for the specific case of a hydrogen-(tetravalent carbon)-hydrogen angle. This hierarchical structure means that a typical parameter file will tend to have generic parameters early in the section for each force type, with more specialized parameters assigned later.

### Multiple SMIRNOFF representations can be processed in sequence

Multiple SMIRNOFF data sources (e.g. multiple OFFXML files) can be loaded by the openforcefield ForceField in sequence. If these files each contain unique top-level tags (such as `<Bonds>`, `<Angles>`, etc.), the resulting forcefield will be independent of the order in which the files are loaded. If, however, the same tag occurs in multiple files, the contents of the tags are merged, with the tags read later taking precedence over the parameters read earlier, provided the top-level tags have compatible attributes. The resulting force field will therefore depend on the order in which parameters are read.

This behavior is intended for limited use in appending very specific parameters, such as parameters specifying solvent models, to override standard parameters.

## 1.3.6 Units

To minimize the potential for [unit conversion errors](#), SMIRNOFF forcefields explicitly specify units in a form readable to both humans and computers for all unit-bearing quantities. Allowed values for units are given in `simtk.unit` (though in the future this may change to the more widely-used Python [pint library](#)). For example, for the angle (equilibrium angle) and k (force constant) parameters in the `<Angle>` example block above, both attributes are specified as a mathematical expression

```
<Angle smirks="[#1:1]-[#6X4:2]-[#1:3]" angle="109.50*degree" k="70.0*kilocalorie_per_mole/radian**2"/>
```

For more information, see the [standard OpenMM unit system](#).

## 1.3.7 SMIRNOFF independently applies parameters to each class of potential energy terms

The SMIRNOFF uses direct chemical perception to assign parameters for potential energy terms independently for each term. Rather than first applying atom typing rules and then looking up combinations of the resulting atom types for each force term, the rules for directly applying parameters to atoms is compartmentalized in separate sections. The file consists of multiple top-level tags defining individual components of the

potential energy (in addition to charge models or modifiers), with each section specifying the typing rules used to assign parameters for that potential term:

```
<Bonds version="0.3" potential="harmonic">
  <Bond smirks="[#6X4:1]-[#6X4:2]" length="1.526*angstrom" k="620.0*kilocalories_per_mole/angstrom**2"/>
  <Bond smirks="[#6X4:1]-[#1:2]" length="1.090*angstrom" k="680.0*kilocalories_per_mole/angstrom**2"/>
  ...
</Bonds>

<Angles version="0.3" potential="harmonic">
  <Angle smirks="[a,A:1]-[#6X4:2]-[a,A:3]" angle="109.50*degree" k="100.0*kilocalories_per_mole/radian**2"/>
  <Angle smirks="[#1:1]-[#6X4:2]-[#1:3]" angle="109.50*degree" k="70.0*kilocalories_per_mole/radian**2"/>
  ...
</Angles>
```

Each top-level tag specifying a class of potential energy terms has an attribute `potential` for specifying the functional form for the interaction. Common defaults are defined, but the goal is to eventually allow these to be overridden by alternative choices or even algebraic expressions in the future, once more molecular simulation packages support general expressions. We distinguish between functional forms available in all common molecular simulation packages (specified by keywords) and support for general functional forms available in a few packages (especially OpenMM, which supports a flexible set of custom forces defined by algebraic expressions) with an **EXPERIMENTAL** label.

Many of the specific forces are implemented as discussed in the [OpenMM Documentation](#); see especially [Section 19 on Standard Forces](#) for mathematical descriptions of these functional forms. Some top-level tags provide attributes that modify the functional form used to be consistent with packages such as AMBER or CHARMM.

### 1.3.8 Partial charge and electrostatics models

SMIRNOFF supports several approaches to specifying electrostatic models. Currently, only classical fixed point charge models are supported, but future extensions to the specification will support point multipoles, point polarizable dipoles, Drude oscillators, charge equilibration methods, and so on.

**<LibraryCharges>: Library charges for polymeric residues and special solvent models**

**Warning:** This functionality is not yet implemented and will appear in a future version of the toolkit. Please see [Issue 25 on the Open Force Field Toolkit issue tracker](#).

A mechanism is provided for specifying library charges that can be applied to molecules or residues that match provided templates. Library charges are applied first, and atoms for which library charges are applied will be excluded from alternative charging schemes listed below.

For example, to assign partial charges for a non-terminal ALA residue from the [AMBER ff14SB](#) parameter set:

```
<LibraryCharges version="0.3">
  <!-- match a non-terminal alanine residue with AMBER ff14SB partial charges-->
  <LibraryCharge name="ALA" smirks="[NX3:1]([#1:2])([#6])([#6H1:3])([#1:4])([#6:5])([#1:6])([#1:7])([#1:8])([#6:9])([#8:10])([#7])" charge1="-0.4157*elementary_charge" charge2="0.2719*elementary_charge"
  charge3="0.0337*elementary_charge" charge4="0.0823*elementary_charge" charge5="-0.1625*elementary_charge" charge6="0.0603*elementary_charge" charge7="0.0603*elementary_charge" charge8="0.
  0603*elementary_charge" charge9="0.5973*elementary_charge" charge10="-0.5679*elementary_charge">
```

(continued from previous page)

```
...
</LibraryCharges>
```

In this case, a SMIRKS string defining the residue tags each atom that should receive a partial charge, with the charges specified by attributes `charge1`, `charge2`, etc. The name attribute is optional. Note that, for a given template, chemically equivalent atoms should be assigned the same charge to avoid undefined behavior. If the template matches multiple non-overlapping sets of atoms, all such matches will be assigned the provided charges. If multiple templates match the same set of atoms, the last template specified will be used.

Solvent models or excipients can also have partial charges specified via the `<LibraryCharges>` tag. For example, to ensure water molecules are assigned partial charges for **TIP3P** water, we can specify a library charge entry:

```
<LibraryCharges version="0.3">
  <!-- TIP3P water oxygen with charge override -->
  <LibraryCharge name="TIP3P" smirks="#1:1]-[#8X2H2+0:2]-[#1:3]" charge1="+0.417*elementary_charge"
  ↪charge2="-0.834*elementary_charge" charge3="+0.417*elementary_charge"/>
</LibraryCharges>
```

#### <ChargeIncrementModel>: Small molecule and fragment charges

**Warning:** This functionality is not yet implemented and will appear in a future version of the toolkit. This area of the SMIRNOFF spec is under further consideration. Please see [Issue 208 on the Open Force Field Toolkit issue tracker](#).

In keeping with the AMBER force field philosophy, especially as implemented in small molecule force fields such as **GAFF**, **GAFF2**, and **parm@Frosst**, partial charges for small molecules are usually assigned using a quantum chemical method (usually a semiempirical method such as **AM1**) and a **partial charge determination scheme** (such as **CM2** or **RESP**), then subsequently corrected via charge increment rules, as in the highly successful **AM1-BCC** approach.

Here is an example:

```
<ChargeIncrementModel version="0.3" number_of_conformers="10" quantum_chemical_method="AM1" partial_
  ↪charge_method="CM2">
  <!-- A fractional charge can be moved along a single bond -->
  <ChargeIncrement smirks="#6X4:1]-[#6X3a:2]" chargeincrement1="-0.0073*elementary_charge"
  ↪chargeincrement2="+0.0073*elementary_charge"/>
  <ChargeIncrement smirks="#6X4:1]-[#6X3a:2]-[#7]" chargeincrement1="+0.0943*elementary_charge"
  ↪chargeincrement2="-0.0943*elementary_charge"/>
  <ChargeIncrement smirks="#6X4:1]-[#8:2]" chargeincrement1="-0.0718*elementary_charge"
  ↪chargeincrement2="+0.0718*elementary_charge"/>
  <!-- Alternatively, factional charges can be redistributed among any number of bonded atoms -->
  <ChargeIncrement smirks="[N:1](H:2)(H:3)" chargeincrement1="+0.02*elementary_charge" chargeincrement2=
  ↪"-0.01*elementary_charge" chargeincrement3="-0.01*elementary_charge"/>
</ChargeIncrementModel>
```

The sum of formal charges for the molecule or fragment will be used to determine the total charge the molecule or fragment will possess.

<ChargeIncrementModel> provides several optional attributes to control its behavior:

- The `number_of_conformers` attribute (default: "10") is used to specify how many conformers will be generated for the molecule (or capped fragment) prior to charging.



- The `quantum_chemical_method` attribute (default: "AM1") is used to specify the quantum chemical method applied to the molecule or capped fragment.
- The `partial_charge_method` attribute (default: "CM2") is used to specify how uncorrected partial charges are to be generated from the quantum chemical wavefunction. Later additions will add restrained electrostatic potential fitting (RESP) capabilities.

The `<ChargeIncrement>` tags specify how the quantum chemical derived charges are to be corrected to produce the final charges. The `chargeincrement#` attributes specify how much the charge on the associated tagged atom index (replacing #) should be modified. The sum of charge increments should equal zero.

Note that atoms for which library charges have already been applied are excluded from charging via `<ChargeIncrementModel>`.

Future additions will provide options for intelligently fragmenting large molecules and biopolymers, as well as a capping attribute to specify how fragments with dangling bonds are to be capped to allow these groups to be charged.

#### `<ToolkitAM1BCC>`: Temporary support for toolkit-based AM1-BCC partial charges

**Warning:** Until `<ChargeIncrementModel>` is implemented, support for the `<ToolkitAM1BCC>` tag has been enabled in the toolkit. This tag is not permanent and may be phased out in future versions of the spec.

This tag calculates partial charges using the default settings of the highest-priority cheminformatics toolkit that can perform [AM1-BCC charge assignment](#). Currently, if the OpenEye toolkit is licensed and available, this will use QuacPac configured to generate charges using [AM1-BCC ELF10](#) for each unique molecule in the topology. Otherwise [RDKit](#) will be used for initial conformer generation and the [AmberTools antechamber/sqm software](#) will be used for charge calculation.

If this tag is specified for a force field, conformer generation will be performed regardless of whether conformations of the input molecule were provided. If [RDKit](#)/[AmberTools](#) are used as the toolkit backend for this calculation, only the first conformer is used for AM1-BCC calculation.

The charges generated by this tag may differ depending on which toolkits are available.

Note that atoms for which prespecified or library charges have already been applied are excluded from charging via `<ToolkitAM1BCC>`.

#### Prespecified charges (reference implementation only)

In our reference implementation of SMIRNOFF in the openforcefield toolkit, we also provide a method for specifying user-defined partial charges during system creation. This functionality is accessed by using the `charge_from_molecules` optional argument during system creation, such as in `ForceField.create_openmm_system(topology, charge_from_molecules=molecule_list)`. When this optional keyword is provided, all matching molecules will have their charges set by the entries in `molecule_list`. This method is provided solely for convenience in developing and exploring alternative charging schemes; actual force field releases for distribution will use one of the other mechanisms specified above.

### 1.3.9 Parameter sections

A SMIRNOFF force field consists of one or more force field term definition sections. For the most part, these sections independently define how a specific component of the potential energy function for a molecular



system is supposed to be computed (such as bond stretch energies, or Lennard-Jones interactions), as well as how parameters are to be assigned for this particular term. Each parameter section contains a version, which encodes the behavior of the section, as well as the required and optional attributes the top-level tag and SMIRKS-based parameters. This decoupling of how parameters are assigned for each term provides a great deal of flexibility in composing new force fields while allowing a minimal number of parameters to be used to achieve accurate modeling of intramolecular forces.

Below, we describe the specification for each force field term definition using the XML representation of a SMIRNOFF force field.

As an example of a complete SMIRNOFF force field specification, see the prototype [SMIRNOFF99Frosst](#) [offxml](#).

**Note:** Not all parameter sections *must* be specified in a SMIRNOFF force field. A wide variety of force field terms are provided in the specification, but a particular force field only needs to define a subset of those terms.

<vdW>

van der Waals force parameters, which include repulsive forces arising from Pauli exclusion and attractive forces arising from dispersion, are specified via the <vdW> tag with sub-tags for individual Atom entries, such as:

```
<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" switch_width="8.0*angstrom" cutoff="9.0*angstrom" long_
range_dispersion="isotropic">
  <Atom smirks="#1:1" sigma="1.4870*angstrom" epsilon="0.0157*kilocalories_per_mole"/>
  <Atom smirks="#1:1-#6" sigma="1.4870*angstrom" epsilon="0.0157*kilocalories_per_mole"/>
  ...
</vdW>
```

For standard Lennard-Jones 12-6 potentials (specified via potential="Lennard-Jones-12-6"), the epsilon parameter denotes the well depth, while the size property can be specified either via providing the sigma attribute, such as sigma="1.3\*angstrom", or via the r\_0/2 (rmin/2) values used in AMBER force fields (here denoted rmin\_half as in the example above). The two are related by  $r_0 = 2^{(1/6)} * \sigma$  and conversion is done internally in ForceField into the sigma values used in OpenMM.

Attributes in the <vdW> tag specify the scaling terms applied to the energies of 1-2 (scale12, default: 0), 1-3 (scale13, default: 0), 1-4 (scale14, default: 0.5), and 1-5 (scale15, default: 1.0) interactions, as well as the distance at which a switching function is applied (switch\_width, default: "1.0\*angstrom"), the cutoff (cutoff, default: "9.0\*angstroms"), and long-range dispersion treatment scheme (long\_range\_dispersion, default: "isotropic").

The potential attribute (default: "none") specifies the potential energy function to use. Currently, only potential="Lennard-Jones-12-6" is supported:

$$U(r) = 4 * \epsilon * ((\sigma/r)^{12} - (\sigma/r)^6)$$

The combining\_rules attribute (default: "none") currently only supports "Lorentz-Berthelot", which specifies the geometric mean of epsilon and arithmetic mean of sigma. Support for [other Lennard-Jones mixing schemes](#) will be added later: Waldman-Hagler, Fender-Halsey, Kong, Tang-Toennies, Pena, Hudson-McCoubrey, Sikora.

Later revisions will add support for additional potential types (e.g., Buckingham-exp-6), as well as the ability to support arbitrary algebraic functional forms using a scheme such as

```
<vdW version="0.3" potential="4*epsilon*((sigma/r)^12-(sigma/r)^6)" scale12="0.0" scale13="0.0" scale14=
↪ "0.5" scale15="1" switch_width="8.0*angstrom" cutoff="9.0*angstrom" long_range_dispersion="isotropic">
  <CombiningRules>
    <CombiningRule parameter="sigma" function="(sigma1+sigma2)/2"/>
    <CombiningRule parameter="epsilon" function="sqrt(epsilon1*epsilon2)"/>
  </CombiningRules>
  <Atom smirks="#1:1" sigma="1.4870*angstrom" epsilon="0.0157*kilocalories_per_mole"/>
  <Atom smirks="#1:1-#6" sigma="1.4870*angstrom" epsilon="0.0157*kilocalories_per_mole"/>
  ...
</vdW>
```

If the <CombiningRules> tag is provided, it overrides the combining\_rules attribute.

Later revisions will also provide support for special interactions using the <AtomPair> tag:

```
<vdW version="0.3" potential="Lennard-Jones-12-6" combining_rules="Lorentz-Berthelot" scale12="0.0"
↪ scale13="0.0" scale14="0.5" scale15="1">
  <AtomPair smirks1="#1:1" smirks2="#6:2" sigma="1.4870*angstrom" epsilon="0.0157*kilocalories_per_
↪ mole"/>
  ...
</vdW>
```

vdW section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	potential="Lennard-Jones-12-6", combining_rules="Lorentz-Berthelot", scale12="0", scale13="0", scale14="0.5", scale15="1.0", cutoff="9.0*angstrom", switch_width="1.0*angstrom", method="cutoff"	smirks, epsilon, (sigma OR rmin_half)	id, parent_id

## <Electrostatics>

Electrostatic interactions are specified via the <Electrostatics> tag.

```
<Electrostatics version="0.3" method="PME" scale12="0.0" scale13="0.0" scale14="0.833333" scale15="1.0"/
↪ >
```

The method attribute specifies the manner in which electrostatic interactions are to be computed:

- PME - **particle mesh Ewald** should be used (DEFAULT); can only apply to periodic systems
- reaction-field - **reaction-field electrostatics** should be used; can only apply to periodic systems
- Coulomb - direct Coulomb interactions (with no reaction-field attenuation) should be used

The interaction scaling parameters applied to atoms connected by a few bonds are

- scale12 (default: 0) specifies the scaling applied to 1-2 bonds
- scale13 (default: 0) specifies the scaling applied to 1-3 bonds
- scale14 (default: 0.833333) specifies the scaling applied to 1-4 bonds
- scale15 (default: 1.0) specifies the scaling applied to 1-5 bonds

Currently, no child tags are used because the charge model is specified via different means (currently library charges or BCCs).

For methods where the cutoff is not simply an implementation detail but determines the potential energy of the system (reaction-field and Coulomb), the cutoff distance must also be specified, and a switch\_width if a switching function is to be used.

Electrostatics section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	scale12="0", scale13="0", scale14="0.833333", scale15="1.0", cutoff="9.0*angstrom", switch_width="0*angstrom", method="PME"	N/A	N/A

### <Bonds>

Bond parameters are specified via a <Bonds>...</Bonds> block, with individual <Bond> tags containing attributes specifying the equilibrium bond length (length) and force constant (k) values for specific bonds. For example:

```
<Bonds version="0.3" potential="harmonic">
  <Bond smirks="#6X4:1]-[#6X4:2]" length="1.526*angstrom" k="620.0*kilocalories_per_mole/angstrom**2"/>
  <Bond smirks="#6X4:1]-[#1:2]" length="1.090*angstrom" k="680.0*kilocalories_per_mole/angstrom**2"/>
  ...
</Bonds>
```

Currently, only potential="harmonic" is supported, where we utilize the standard harmonic functional form:

$$U(r) = (k/2)*(r-\text{length})^2$$

Later revisions will add support for additional potential types and the ability to support arbitrary algebraic functional forms. If the potential attribute is omitted, it defaults to harmonic.

**Note that AMBER and CHARMM define a modified functional form**, such that  $U(r) = k*(r-\text{length})^2$ , so that force constants would need to be multiplied by two in order to be used in the SMIRNOFF format.

Constrained bonds are handled by a separate <Constraints> tag, which can either specify constraint distances or draw them from equilibrium distances specified in <Bonds>.

### Fractional bond orders (EXPERIMENTAL)

**Warning:** This functionality is not yet implemented and will appear in a future version of the toolkit.

Fractional bond orders can be used to allow interpolation of bond parameters. For example, these parameters:

```
<Bonds version="0.3" potential="harmonic">
  <Bond smirks="#6X3:1]-[#6X3:2]" k="820.0*kilocalories_per_mole/angstrom**2" length="1.45*angstrom"/>
  <Bond smirks="#6X3:1]:[#6X3:2]" k="938.0*kilocalories_per_mole/angstrom**2" length="1.40*angstrom"/>
  <Bond smirks="#6X3:1]=[#6X3:2]" k="1098.0*kilocalories_per_mole/angstrom**2" length="1.35*angstrom"/>
  ...
</Bonds>
```

can be replaced by a single parameter line by first invoking the `fractional_bondorder_method` attribute to specify a method for computing the fractional bond order and `fractional_bondorder_interpolation` for specifying the procedure for interpolating parameters between specified integral bond orders:

```
<Bonds version="0.3" potential="harmonic" fractional_bondorder_method="Wiberg" fractional_bondorder_
↪interpolation="linear">
  <Bond smirks="#6X3:1]!#[6X3:2]" k_bondorder1="820.0*kilocalories_per_mole/angstrom**2" k_
↪bondorder2="1098*kilocalories_per_mole/angstrom**2" length_bondorder1="1.45*angstrom" length_
↪bondorder2="1.35*angstrom"/>
  ...
```

This allows specification of force constants and lengths for bond orders 1 and 2, and then interpolation between those based on the partial bond order.

- `fractional_bondorder_method` defaults to none, but the Wiberg method is supported.
- `fractional_bondorder_interpolation` defaults to linear, which is the only supported scheme for now.

Bonds section version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	potential="harmonic", fractional_bondorder_method="none", fractional_bondorder_interpolation="linear"	smirks, length, k	id, parent_id

### <Angles>

Angle parameters are specified via an `<Angles>...</Angles>` block, with individual `<Angle>` tags containing attributes specifying the equilibrium angle (`angle`) and force constant (`k`), as in this example:

```
<Angles version="0.3" potential="harmonic">
  <Angle smirks="[a,A:1]-[#6X4:2]-[a,A:3]" angle="109.50*degree" k="100.0*kilocalories_per_mole/
↪radian**2"/>
  <Angle smirks="#1:1]-[#6X4:2]-[#1:3]" angle="109.50*degree" k="70.0*kilocalories_per_mole/radian**2
↪"/>
  ...
</Angles>
```

Currently, only `potential="harmonic"` is supported, where we utilize the standard harmonic functional form:

$$U(r) = (k/2) * (\text{theta} - \text{angle})^2$$

Later revisions will add support for additional potential types and the ability to support arbitrary algebraic functional forms. If the potential attribute is omitted, it defaults to harmonic.

**Note that AMBER and CHARMM define a modified functional form**, such that  $U(r) = k * (\text{theta} - \text{angle})^2$ , so that force constants would need to be multiplied by two in order to be used in the SMIRNOFF format.

Angles section version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	potential="harmonic"	smirks, angle, k	id, parent_id

### <ProperTorsions>

Proper torsions are specified via a `<ProperTorsions>...</ProperTorsions>` block, with individual `<Proper>` tags containing attributes specifying the periodicity (`periodicity#`), phase (`phase#`), and barrier height (`k#`).

```
<ProperTorsions version="0.3" potential="k*(1+cos(periodicity*theta-phase))">
  <Proper smirks="[a,A:1]-[#6X4:2]-[#6X4:3]-[a,A:4]" idivf1="9" periodicity1="3" phase1="0.0*degree"
  ↪k1="1.40*kilocalories_per_mole"/>
  <Proper smirks="[#6X4:1]-[#6X4:2]-[#8X2:3]-[#6X4:4]" idivf1="1" periodicity1="3" phase1="0.0*degree"
  ↪k1="0.383*kilocalories_per_mole" idivf2="1" periodicity2="2" phase2="180.0*degree" k2="0.
  ↪1*kilocalories_per_mole"/>
  ...
</ProperTorsions>
```

Here, child Proper tags specify at least  $k_1$ ,  $\text{phase}_1$ , and  $\text{periodicity}_1$  attributes for the corresponding parameters of the first force term applied to this torsion. However, additional values are allowed in the form  $k\#$ ,  $\text{phase}\#$ , and  $\text{periodicity}\#$ , where all  $\#$  values must be consecutive (e.g., it is impermissible to specify  $k_1$  and  $k_3$  values without a  $k_2$  value) but  $\#$  can go as high as necessary.

For convenience, an optional attribute specifies a torsion multiplicity by which the barrier height should be divided ( $\text{idivf}\#$ ). The default behavior of this attribute can be controlled by the top-level attribute `default_idivf` (default: "auto") for `<ProperTorsions>`, which can be an integer (such as "1") controlling the value of  $\text{idivf}$  if not specified or "auto" if the barrier height should be divided by the number of torsions impinging on the central bond. For example:

```
<ProperTorsions version="0.3" potential="k*(1+cos(periodicity*theta-phase))" default_idivf="auto">
  <Proper smirks="[a,A:1]-[#6X4:2]-[#6X4:3]-[a,A:4]" periodicity1="3" phase1="0.0*degree" k1="1.
  ↪40*kilocalories_per_mole"/>
  ...
</ProperTorsions>
```

Currently, only  $\text{potential}="k*(1+\cos(\text{periodicity}*\theta-\text{phase}))"$  is supported, where we utilize the functional form:

$$U = \sum_{i=1}^N k_i * (1 + \cos(\text{periodicity}_i * \phi - \text{phase}_i))$$

**Note:** AMBER defines a modified functional form, such that  $U = \sum_{i=1}^N (k_i/2) * (1 + \cos(\text{periodicity}_i * \phi - \text{phase}_i))$ , so that barrier heights would need to be divided by two in order to be used in the SMIRNOFF format.

If the potential attribute is omitted, it defaults to  $k*(1+\cos(\text{periodicity}*\theta-\text{phase}))$ .

ProperTorsions section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	<code>potential="k*(1+cos(periodicity*theta-phase))"</code> <code>default_idivf="auto"</code>	<code>k</code> , <code>k</code> , <code>phase</code> , <code>periodicity</code>	<code>idivf</code> , <code>id</code> , <code>parent_id</code>

### <ImproperTorsions>

Improper torsions are specified via an `<ImproperTorsions>...</ImproperTorsions>` block, with individual `<Improper>` tags containing attributes that specify the same properties as `<ProperTorsions>`:

```
<ImproperTorsions version="0.3" potential="k*(1+cos(periodicity*theta-phase))">
  <Improper smirks="[*:1]~[#6X3:2](=[#7X2,#7X3+1:3])~[#7:4]" k1="10.5*kilocalories_per_mole"
  ↪periodicity1="2" phase1="180.*degree"/>
  ...
</ImproperTorsions>
```

Currently, only `potential="charmm"` is supported, where we utilize the functional form:

$$U = \sum_{i=1}^N k_i * (1 + \cos(\text{periodicity}_i * \phi_i - \text{phase}_i))$$

**Note:** AMBER defines a modified functional form, such that  $U = \sum_{i=1}^N (k_i/2) * (1 + \cos(\text{periodicity}_i * \phi_i - \text{phase}_i))$ , so that barrier heights would need to be divided by two in order to be used in the SMIRNOFF format.

If the potential attribute is omitted, it defaults to `charmm`.

The improper torsion energy is computed as the average over all three impropers (all with the same handedness) in a [trefoil](#). This avoids the dependence on arbitrary atom orderings that occur in more traditional typing engines such as those used in AMBER. The *second* atom in an improper (in the example above, the trivalent carbon) is the central atom in the trefoil.

ImproperTorsions section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	<code>potential="k*(1+cos(periodicity*theta-phase))"</code> , <code>default_idivf="auto"</code>	<code>k</code> , <code>phase</code> , <code>periodicity</code>	<code>idivf</code> , <code>id</code> , <code>parent_id</code>

<GBSA>

**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.

Generalized-Born surface area (GBSA) implicit solvent parameters are optionally specified via a `<GBSA>...` `</GBSA>` using `<Atom>` tags with GBSA model specific attributes:

```
<GBSA version="0.3" gb_model="OBC1" solvent_dielectric="78.5" solute_dielectric="1" sa_model="ACE"
surface_area_penalty="5.4*calories/mole/angstroms**2" solvent_radius="1.4*angstroms">
  <Atom smirks="[*:1]" radius="0.15*nanometer" scale="0.8"/>
  <Atom smirks="[#1:1]" radius="0.12*nanometer" scale="0.85"/>
  <Atom smirks="[#1:1]~[#7]" radius="0.13*nanometer" scale="0.85"/>
  <Atom smirks="[#6:1]" radius="0.17*nanometer" scale="0.72"/>
  <Atom smirks="[#7:1]" radius="0.155*nanometer" scale="0.79"/>
  <Atom smirks="[#8:1]" radius="0.15*nanometer" scale="0.85"/>
  <Atom smirks="[#9:1]" radius="0.15*nanometer" scale="0.88"/>
  <Atom smirks="[#14:1]" radius="0.21*nanometer" scale="0.8"/>
  <Atom smirks="[#15:1]" radius="0.185*nanometer" scale="0.86"/>
  <Atom smirks="[#16:1]" radius="0.18*nanometer" scale="0.96"/>
  <Atom smirks="[#17:1]" radius="0.17*nanometer" scale="0.8"/>
</GBSA>
```

## Supported Generalized Born (GB) models

In the `<GBSA>` tag, `gb_model` selects which GB model is used. Currently, this can be selected from a subset of the GBSA models available in [OpenMM](#):

- HCT : [Hawkins-Cramer-Truhlar](#) (corresponding to `igb=1` in AMBER): requires parameters [`radius`, `scale`]

- OBC1 : [Onufriev-Bashford-Case](#) using the GB(OBC)I parameters (corresponding to igb=2 in AMBER): requires parameters [radius, scale]
- OBC2 : [Onufriev-Bashford-Case](#) using the GB(OBC)II parameters (corresponding to igb=5 in AMBER): requires parameters [radius, scale]

If the gb\_model attribute is omitted, it defaults to OBC1.

The attributes solvent\_dielectric and solute\_dielectric specify solvent and solute dielectric constants used by the GB model. In this example, radius and scale are per-particle parameters of the OBC1 GB model supported by OpenMM.

### Surface area (SA) penalty model

The sa\_model attribute specifies the solvent-accessible surface area model (“SA” part of GBSA) if one should be included; if omitted, no SA term is included.

Currently, only the [analytical continuum electrostatics \(ACE\) model](#), designated ACE, can be specified, but there are plans to add more models in the future, such as the Gaussian solvation energy component of [EEF1](#). If sa\_model is not specified, it defaults to ACE.

The ACE model permits two additional parameters to be specified:

- The surface\_area\_penalty attribute specifies the surface area penalty for the ACE model. (Default: 5.4 calories/mole/angstroms\*\*2)
- The solvent\_radius attribute specifies the solvent radius. (Default: 1.4 angstroms)

GBSA section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	gb_model="OBC1", solvent_dielectric="78.5", solute_dielectric="1", sa_model="ACE", surface_area_penalty="5.4*calories/mole/angstrom**2", solvent_radius="1.4*angstrom"	smirks, radius, scale	id, parent_id

### <Constraints>

Bond length or angle constraints can be specified through a <Constraints> block, which can constrain bonds to their equilibrium lengths or specify an interatomic constraint distance. Two atoms must be tagged in the smirks attribute of each <Constraint> record.

To constrain the separation between two atoms to their equilibrium bond length, it is critical that a <Bonds> record be specified for those atoms:

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

Note that the two atoms must be bonded in the specified Topology for the equilibrium bond length to be used.

To specify the constraint distance, or constrain two atoms that are not directly bonded (such as the hydrogens in rigid water models), specify the distance attribute (and optional distance\_unit attribute for the <Constraints> tag):

```
<Constraints version="0.3">
  <!-- constrain water O-H bond to equilibrium bond length (overrides earlier constraint) -->
  <Constraint smirks="#1:1]-[#8X2H2:2]-[#1]" distance="0.9572*angstrom"/>
  <!-- constrain water H...H, calculating equilibrium length from H-O-H equilibrium angle and H-O_
  equilibrium bond lengths -->
  <Constraint smirks="#1:1]-[#8X2H2]-[#1:2]" distance="1.8532*angstrom"/>
</Constraints>
```

Typical molecular simulation practice is to constrain all bonds to hydrogen to their equilibrium bond lengths and enforce rigid TIP3P geometry on water molecules:

```
<Constraints version="0.3">
  <!-- constrain all bonds to hydrogen to their equilibrium bond length -->
  <Constraint smirks="#1:1]-[*:2]" />
  <!-- TIP3P rigid water -->
  <Constraint smirks="#1:1]-[#8X2H2:2]-[#1]" distance="0.9572*angstrom"/>
  <Constraint smirks="#1:1]-[#8X2H2]-[#1:2]" distance="1.8532*angstrom"/>
</Constraints>
```

Constraint section tag version	Required tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3		smirks	distance

### 1.3.10 Advanced features

Standard usage is expected to rely primarily on the features documented above and potentially new features. However, some advanced features will also be supported.

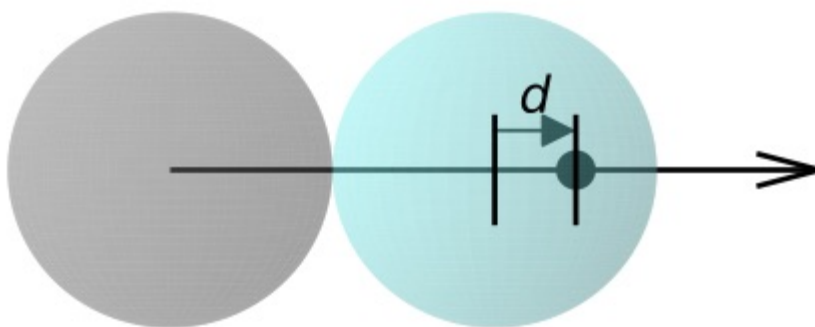
#### <VirtualSites>: Virtual sites for off-atom charges

**Warning:** This functionality is not yet implemented and will appear in a future version of the toolkit

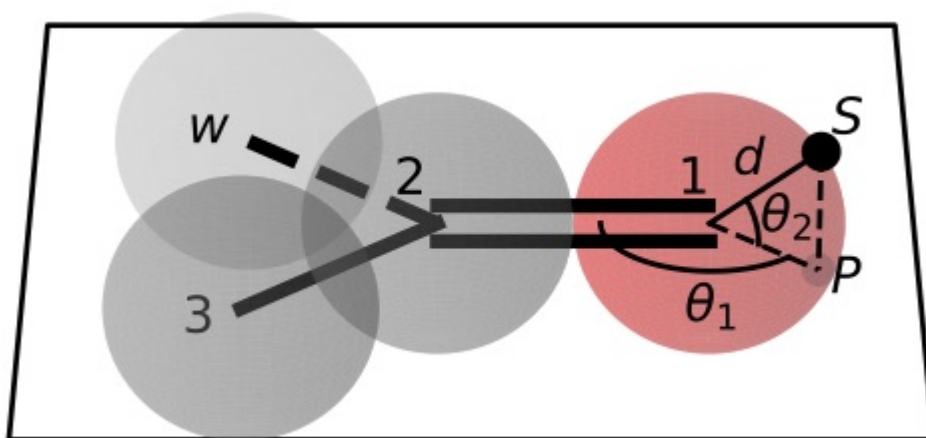
We will implement experimental support for placement of off-atom (off-center) charges in a variety of contexts which may be chemically important in order to allow easy exploration of when these will be warranted. We will support the following different types or geometries of off-center charges (as diagrammed below):

- **BondCharge:** This supports placement of a virtual site S along a vector between two specified atoms, e.g. to allow for a sigma hole for halogens or similar contexts. With positive values of the distance, the virtual site lies outside the first indexed atom (green in this image).

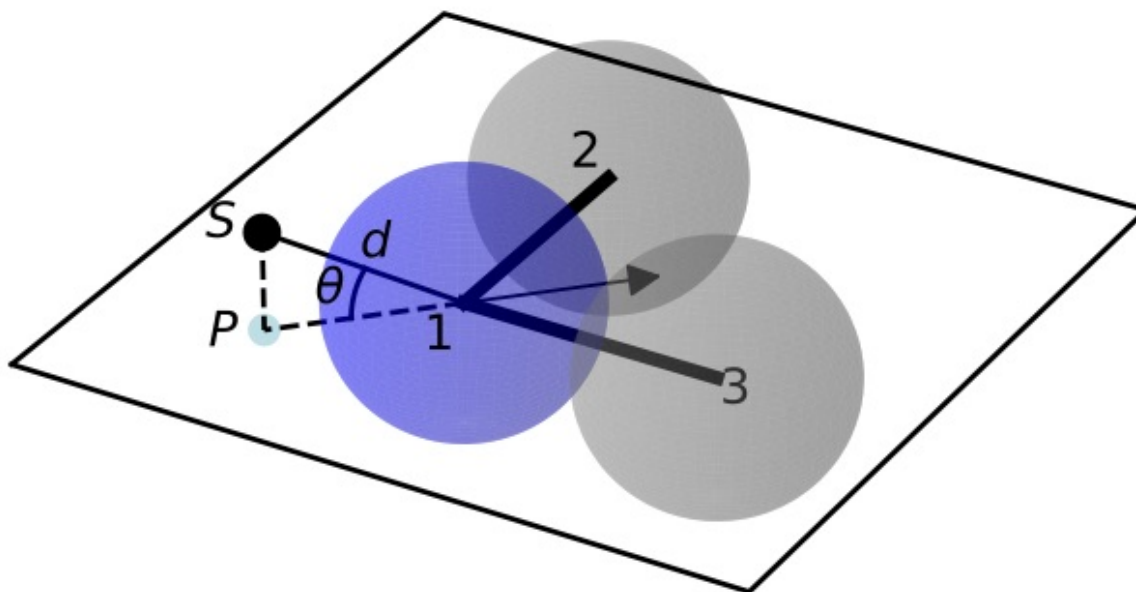




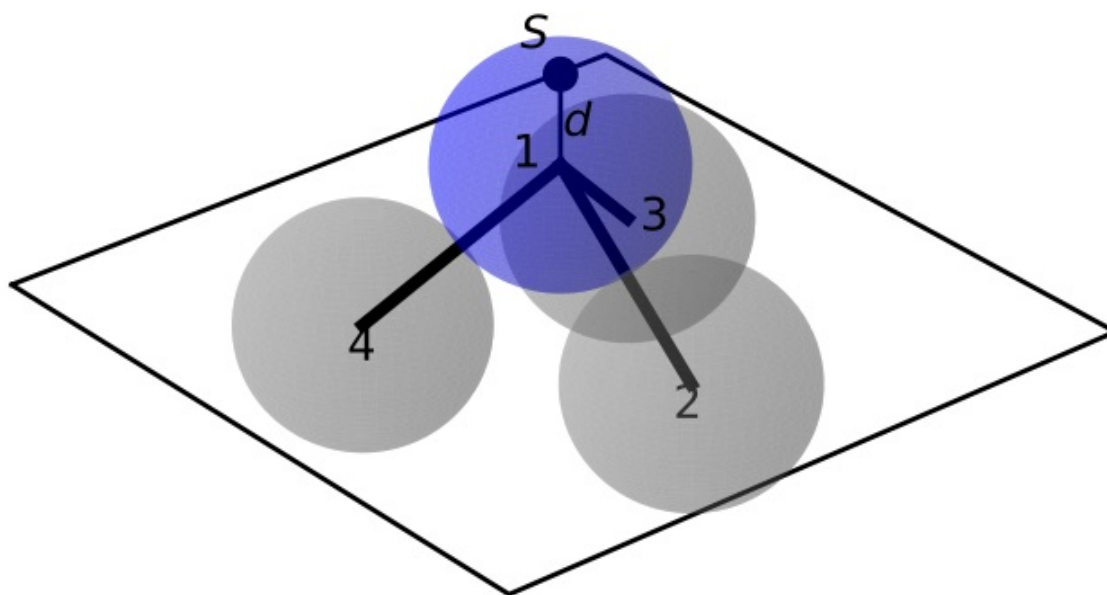
- **MonovalentLonePair:** This is originally intended for situations like a carbonyl, and allows placement of a virtual site  $S$  at a specified distance  $d$ , `inPlaneAngle` (theta 1 in the diagram), and `outOfPlaneAngle` (theta 2 in the diagram) relative to a central atom and two connected atoms.



- **DivalentLonePair:** This is suitable for cases like four-point and five-point water models as well as pyrimidine; a charge site  $S$  lies a specified distance  $d$  from the central atom among three atoms (blue) along the bisector of the angle between the atoms (if `outOfPlaneAngle` is zero) or out of the plane by the specified angle (if `outOfPlaneAngle` is nonzero) with its projection along the bisector. For positive values for the distance  $d$  the virtual site lies outside the 2-1-3 angle and for negative values it lies inside.



- **TrivalentLonePair:** This is suitable for planar or tetrahedral nitrogen lone pairs; a charge site *S* lies above the central atom (e.g. nitrogen, blue) a distance *d* along the vector perpendicular to the plane of the three connected atoms (2,3,4). With positive values of *d* the site lies above the nitrogen and with negative values it lies below the nitrogen.



Each virtual site receives charge which is transferred from the desired atoms specified in the SMIRKS pattern via a `chargeincrement#` parameter; e.g., if `chargeincrement1=+0.1*elementary_charge` then the virtual site will receive a charge of -0.1 and the atom labeled 1 will have its charge adjusted upwards by +0.1. N may index any indexed atom. Increments which are left unspecified default to zero. Additionally, each virtual site can bear Lennard-Jones parameters, specified by `sigma` and `epsilon` or `rmin_half` and `epsilon`. If unspecified these also default to zero.

In the SMIRNOFF format, these are encoded as:

```

<VirtualSites version="0.3">
  <!-- sigma hole for halogens: "distance" denotes distance along the 2->1 bond vector, measured from_
  ↪ atom 2 -->
  <!-- Specify that 0.2 charge from atom 1 and 0.1 charge units from atom 2 are to be moved to the_
  ↪ virtual site, and a small Lennard-Jones site is to be added (sigma = 0.1*angstroms, epsilon=0.05*kcal/_
  ↪ mol) -->
  <VirtualSite type="BondCharge" smirks="[C1:1]-[C:2]" distance="0.30*angstrom" chargeincrement1="+0.
  ↪ 2*elementary_charge" chargeincrement2="+0.1*elementary_charge" sigma="0.1*angstrom" epsilon="0.
  ↪ 05*kilocalories_per_mole" />
  <!-- Charge increments can extend out to as many atoms as are labeled, e.g. with a third atom: -->
  <VirtualSite type="BondCharge" smirks="[C1:1]-[C:2]~[*:3]" distance="0.30*angstrom"
  ↪ chargeincrement1="+0.1*elementary_charge" chargeincrement2="+0.1*elementary_charge" chargeincrement3=
  ↪ "+0.05*elementary_charge" sigma="0.1*angstrom" epsilon="0.05*kilocalories_per_mole" />
  <!-- monovalent lone pairs: carbonyl -->
  <!-- X denotes the charge site, and P denotes the projection of the charge site into the plane of 1_
  ↪ and 2. -->
  <!-- inPlaneAngle is angle point P makes with 1 and 2, i.e. P-1-2 -->
  <!-- outOfPlaneAngle is angle charge site (X) makes out of the plane of 2-1-3 (and P) measured from_
  ↪ 1 -->
  <!-- Since unspecified here, sigma and epsilon for the virtual site default to zero -->
  <VirtualSite type="MonovalentLonePair" smirks="[O:1]=[C:2]-[*:3]" distance="0.30*angstrom"
  ↪ outOfPlaneAngle="0*degree" inPlaneAngle="120*degree" chargeincrement1="+0.2*elementary_charge" />
  <!-- divalent lone pair: pyrimidine, TIP4P, TIP5P -->
  <!-- The atoms 2-1-3 define the X-Y plane, with Z perpendicular. If outOfPlaneAngle is 0, the_
  ↪ charge site is a specified distance along the in-plane vector which bisects the angle left by taking_
  ↪ 360 degrees minus angle(2,1,3). If outOfPlaneAngle is nonzero, the charge sites lie out of the plane_
  ↪ by the specified angle (at the specified distance) and their in-plane projection lines along the angle_
  ↪ 's bisector. -->
  <VirtualSite type="DivalentLonePair" smirks="[*:2]~[#7X2:1]~[*:3]" distance="0.30*angstrom"
  ↪ outOfPlaneAngle="0.0*degree" chargeincrement1="+0.1*elementary_charge" />
  <!-- trivalent nitrogen lone pair -->
  <!-- charge sites lie above and below the nitrogen at specified distances from the nitrogen, along_
  ↪ the vector perpendicular to the plane of (2,3,4) that passes through the nitrogen. If the nitrogen is_
  ↪ co-planar with the connected atom, charge sites are simply above and below the plane-->
  <!-- Positive and negative values refer to above or below the nitrogen as measured relative to the_
  ↪ plane of (2,3,4), i.e. below the nitrogen means nearer the 2,3,4 plane unless they are co-planar -->
  <VirtualSite type="TrivalentLonePair" smirks="[*:2]~[#7X3:1](~[*:4])~[*:3]" distance="0.30*angstrom
  ↪ " chargeincrement1="+0.1*elementary_charge"/>
  <VirtualSite type="TrivalentLonePair" smirks="[*:2]~[#7X3:1](~[*:4])~[*:3]" distance="-0.30*angstrom
  ↪ " chargeincrement1="+0.1*elementary_charge"/>
</VirtualSites>

```

## Aromaticity models

Before conducting SMIRKS substructure searches, molecules are prepared using one of the supported aromaticity models, which must be specified with the `aromaticity_model` attribute. The only aromaticity model currently widely supported (by both the [OpenEye toolkit](#) and [RDKit](#)) is the `OEArModel_MDL` model.

## Additional plans for future development

See the [openforcefield GitHub issue tracker](#) to propose changes to this specification, or read through proposed changes currently being discussed.

### 1.3.11 The openforcefield reference implementation

A Python reference implementation of a parameterization engine implementing the SMIRNOFF force field specification can be found [online](#). This implementation can use either the free-for-academics (but commercially supported) [OpenEye toolkit](#) or the free and open source [RDKit cheminformatics toolkit](#). See the [installation instructions](#) for information on how to install this implementation and its dependencies.

#### Examples

A relatively extensive set of examples is made available on the [reference implementation repository](#) under [examples/](#).

#### Parameterizing a system

Consider parameterizing a simple system containing a the drug imatinib.

```
# Create a molecule from a mol2 file
from openforcefield.topology import Molecule
molecule = Molecule.from_file('imatinib.mol2')

# Create a Topology specifying the system to be parameterized containing just the molecule
topology = molecule.to_topology()

# Load the smirnoff99Frosst forcefield
from openforcefield.typing.engines import smirnoff
forcefield = smirnoff.ForceField('test_forcefields/smirnoff99Frosst.offxml')

# Create an OpenMM System from the topology
system = forcefield.create_openmm_system(topology)
```

See [examples/SMIRNOFF\\_simulation/](#) for an extension of this example illustrating to simulate this molecule in the gas phase.

The topology object provided to `create_openmm_system()` can contain any number of molecules of different types, including biopolymers, ions, buffer molecules, or solvent molecules. The openforcefield toolkit provides a number of convenient methods for importing or constructing topologies given PDB files, Sybyl mol2 files, SDF files, SMILES strings, and IUPAC names; see the [toolkit documentation](#) for more information. Notably, this topology object differs from those found in [OpenMM](#) or [MDTraj](#) in that it contains information on the *chemical identity* of the molecules constituting the system, rather than this atomic elements and covalent connectivity; this additional chemical information is required for the [direct chemical perception](#) features of SMIRNOFF typing.

#### Using SMIRNOFF small molecule forcefields with traditional biopolymer force fields

While SMIRNOFF format force fields can cover a wide range of biological systems, our initial focus is on general small molecule force fields, meaning that users may have considerable interest in combining SMIRNOFF small molecule parameters to systems in combination with traditional biopolymer parameters from conventional force fields, such as the AMBER family of protein/nucleic acid force fields. Thus, we provide an example of setting up a mixed protein-ligand system in [examples/mixedFF\\_structure](#), where an AMBER family force field is used for a protein and smirnoff99Frosst for a small molecule.

### The optional `id` and `parent_id` attributes and other XML attributes

In general, additional optional XML attributes can be specified and will be ignored by ForceField unless they are specifically handled by the parser (and specified in this document).

One attribute we have found helpful in parameter file development is the `id` attribute for a specific parameter line, and we *recommend* that SMIRNOFF force fields utilize this as effectively a parameter serial number, such as in:

```
<Bond smirks="#6X3:1]-[#6X3:2]" id="b5" k="820.0*kilocalorie_per_mole/angstrom**2" length="1.45*angstrom"/>
```

Some functionality in ForceField, such as `ForceField.label_molecules`, looks for the `id` attribute. Without this attribute, there is no way to uniquely identify a specific parameter line in the XML file without referring to it by its smirks string, and since some smirks strings can become long and relatively unwieldy (especially for torsions) this provides a more human- and search-friendly way of referring to specific sets of parameters.

The `parent_id` attribute is also frequently used to denote parameters from which the current parameter is derived in some manner.

### A remark about parameter availability

ForceField will currently raise an exception if any parameters are missing where expected for your system—i.e. if a bond is assigned no parameters, an exception will be raised. However, use of generic parameters (i.e. `[*:1]~[:2]` for a bond) in your `.offxml` will result in parameters being assigned everywhere, bypassing this exception. We recommend generics be used sparingly unless it is your intention to provide true universal generic parameters.

## 1.3.12 Version history

### 0.3

This is a backwards-incompatible update to the SMIRNOFF 0.2 draft specification. However, the Open Force Field Toolkit version accompanying this update is capable of converting 0.1 spec SMIRNOFF data to 0.2 spec, and subsequently 0.2 spec to 0.3 spec. The 0.1-to-0.2 spec conversion makes a number of assumptions about settings such as long-range nonbonded handling. Warnings are printed about each assumption that is made during this spec conversion. No mechanism to convert backwards in spec is provided.

Key changes in this version of the spec are:

- Section headers now contain individual versions, instead of relying on the `<SMIRNOFF>`-level tag.
- Section headers no longer contain `X_unit` attributes.
- All physical quantities are now written as expressions containing the appropriate units.
- The default potential for `<ProperTorsions>` and `<ImproperTorsions>` was changed from `charmm` to `k*(1+cos(periodicity*theta-phase))`, as CHARMM interprets torsion terms with periodicity 0 as having a quadratic potential, while the Open Force Field Toolkit would interpret a zero periodicity literally.

### 0.2

This is a backwards-incompatible overhaul of the SMIRNOFF 0.1 draft specification along with ForceField implementation refactor:

- Aromaticity model now defaults to `OEArModel_MDL`, and aromaticity model names drop OpenEye-specific prefixes
- Top-level tags are now required to specify units for any unit-bearing quantities to avoid the potential for mistakes from implied units.
- Potential energy component definitions were renamed to be more general:
  - `<NonbondedForce>` was renamed to `<vdW>`
  - `<HarmonicBondForce>` was renamed to `<Bonds>`
  - `<HarmonicAngleForce>` was renamed to `<Angles>`
  - `<BondChargeCorrections>` was renamed to `<ChargeIncrementModel>` and generalized to accommodate an arbitrary number of tagged atoms
  - `<GBSAForce>` was renamed to `<GBSA>`
- `<PeriodicTorsionForce>` was split into `<ProperTorsions>` and `<ImproperTorsions>`
- `<vdW>` now specifies 1-2, 1-3, 1-4, and 1-5 scaling factors via `scale12` (default: 0), `scale13` (default: 0), `scale14` (default: 0.5), and `scale15` (default 1.0) attributes. It also specifies the long-range vdW method to use, currently supporting cutoff (default) and PME. Coulomb scaling parameters have been removed from `StericsForce`.
- Added the `<Electrostatics>` tag to separately specify 1-2, 1-3, 1-4, and 1-5 scaling factors for electrostatics, as well as the method used to compute electrostatics (PME, reaction-field, Coulomb) since this has a huge effect on the energetics of the system.
- Made it clear that `<Constraint>` entries do not have to be between bonded atoms.
- `<VirtualSites>` has been added, and the specification of charge increments harmonized with `<ChargeIncrementModel>`
- The potential attribute was added to most forces to allow flexibility in extending forces to additional functional forms (or algebraic expressions) in the future. potential defaults to the current recommended scheme if omitted.
- `<GBSA>` now has defaults specified for `gb_method` and `sa_method`
- Changes to how fractional bond orders are handled:
  - Use of fractional bond order is now are specified at the force tag level, rather than the root level
  - The fractional bond order method is specified via the `fractional_bondorder_method` attribute
  - The fractional bond order interpolation scheme is specified via the `fractional_bondorder_interpolation`
- Section heading names were cleaned up.
- Example was updated to reflect use of the new `openforcefield.topology.Topology` class
- Eliminated “Requirements” section, since it specified requirements for the software, rather than described an aspect of the SMIRNOFF specification
- Fractional bond orders are described in `<Bonds>`, since they currently only apply to this term.

## 0.1

Initial draft specification.

## 1.4 Examples using SMIRNOFF with the toolkit

The following examples are available in [the openforcefield toolkit repository](#):

### 1.4.1 Index of provided examples

- [SMIRNOFF\\_simulation](#) - simulation of a molecule in the gas phase with the SMIRNOFF forcefield format
- [forcefield\\_modification](#) - modify forcefield parameters and evaluate how system energy changes
- [using\\_smirnoff\\_in\\_amber\\_or\\_gromacs](#) - convert a System generated with the Open Forcefield Toolkit, which can be simulated natively with OpenMM, into AMBER prmtop/inpcrd and GROMACS top/gro input files through the ParmEd library.
- [inspect\\_assigned\\_parameters](#) - check which parameters are used in which molecules and generate parameter usage statistics.
- [using\\_smirnoff\\_with\\_amber\\_protein\\_forcefield](#) - use SMIRNOFF parameters for small molecules in combination with more conventional force fields for proteins and other components of your system (using ParmEd to combine parameterized structures)
- [check\\_dataset\\_parameter\\_coverage](#) - shows how to use the Open Force Field Toolkit to ingest a dataset of molecules, and generate a report summarizing any chemistry that can not be parameterized.

## 1.5 Developing for the toolkit

### 1.5.1 Style guide

Development for the openforcefield 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, consistently with the variables in the standard `pathlib` library. - Use `n_x` to abbreviate “number of X” (e.g. `n_atoms`, `n_molecules`).

### 1.5.2 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.

### 1.5.3 How can I become a developer?

If you would like to contribute, please post an issue on the [GitHub issue tracker](#) describing the contribution you would like to make to start a discussion.



## 1.6 Frequently asked questions (FAQ)

### 1.6.1 Input files for applying SMIRNOFF parameters

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.

### 1.6.2 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.

If you have such an inference problem, we recommend that you use pre-existing cheminformatics tools available elsewhere (such as via the OpenEye toolkits, such as the `OEPerceiveBondOrders` functionality offered there) to solve this problem and identify your molecules before beginning your work with SMIRNOFF.

### 1.6.3 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 above) 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

#### 1.6.4 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 .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 .mol2 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.



## API DOCUMENTATION

## 2.1 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](#)).

### 2.1.1 Primary objects

<code>FrozenMolecule</code>	Immutable chemical representation of a molecule, such as a small molecule or biopolymer.
<code>Molecule</code>	Mutable chemical representation of a molecule, such as a small molecule or biopolymer.
<code>Topology</code>	A Topology is a chemical representation of a system containing one or more molecules appearing in a specified order.
<code>TopologyMolecule</code>	TopologyMolecules are built to be an efficient way to store large numbers of copies of the same molecule for parameterization and system preparation.

#### `openforcefield.topology.FrozenMolecule`

```
class openforcefield.topology.FrozenMolecule(other=None, file_format=None,  
                                              toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry  
                                              object>, allow_undefined_stereo=False)
```

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

#### Examples

Create a molecule from a sdf file

```
>>> from openforcefield.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.

### Attributes

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

**atoms** Iterate over all Atom objects.

**bonds** Iterate over all Bond objects.

**conformers** Returns the list of conformers for this molecule.

**impropers** Iterate over all proper torsions in the molecule

**n\_angles** int: number of angles in the Molecule.

**n\_atoms** The number of Atom objects.

**n\_bonds** The number of Bond objects.

**n\_conformers** Returns the number of conformers for this molecule.

**n\_impropers** int: number of improper torsions in the Molecule.

**n\_particles** The number of Particle objects, which corresponds to how many positions must be used.

**n\_propers** int: number of proper torsions in the Molecule.

**n\_virtual\_sites** The number of VirtualSite objects.

**name** The name (or title) of the molecule

**partial\_charges** Returns the partial charges (if present) on the molecule.

**particles** Iterate over all Particle objects.

**propers** Iterate over all proper torsions in the molecule

**properties** The properties dictionary of the molecule

**torsions** Get an iterator over all i-j-k-l torsions.

**total\_charge** Return the total charge on the molecule

**virtual\_sites** Iterate over all VirtualSite objects.

## Methods

<code>chemical_environment_matches(self, query[, ...])</code>		Retrieve all matches for a given chemical environment query.
<code>compute_partial_charges(self, toolkit_registry)</code>		<b>Warning! Not Implemented!</b> Calculate partial atomic charges for this molecule using an underlying toolkit
<code>compute_partial_charges_am1bcc(self, ...)</code>		Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit
<code>compute_wiberg_bond_orders(self, ...)</code>		Calculate wiberg bond orders for this molecule using an underlying toolkit
<code>from_bson(serialized)</code>		Instantiate an object from a BSON serialized representation.
<code>from_dict(molecule_dict)</code>		Create a new Molecule from a dictionary representation
<code>from_file(file_path[, file_format, ...])</code>		Create one or more molecules from a file
<code>from_iupac(iupac_name, **kwargs)</code>		Generate a molecule from IUPAC or common name
<code>from_json(serialized)</code>		Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>		Instantiate an object from a MessagePack serialized representation.
<code>from_openeye(oemol[, allow_undefined_stereo])</code>	al-	Create a Molecule from an OpenEye molecule.
<code>from_pickle(serialized)</code>		Instantiate an object from a pickle serialized representation.
<code>from_rdkit(rdmol[, allow_undefined_stereo])</code>		Create a Molecule from an RDKit molecule.
<code>from_smiles(smiles[, ...])</code>		Construct a Molecule from a SMILES representation
<code>from_toml(serialized)</code>		Instantiate an object from a TOML serialized representation.
<code>from_topology(topology)</code>		Return a Molecule representation of an openforcefield Topology containing a single Molecule object.
<code>from_xml(serialized)</code>		Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>		Instantiate from a YAML serialized representation.
<code>generate_conformers(self, ...)</code>		Generate conformers for this molecule using an underlying toolkit
<code>get_bond_between(self, i, j)</code>		Returns the bond between two atoms
<code>get_fractional_bond_orders(self, method, ...)</code>		Get fractional bond orders.
<code>is_isomorphic(self, other[, ...])</code>		Determines whether the molecules are isomorphic by comparing their graphs.

Continued on next page

Table 2 – continued from previous page

<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Return a dictionary representation of the molecule.
<code>to_file(self, file_path, file_format[, ...])</code>	Write the current molecule to a file or file-like object
<code>to_iupac(self)</code>	Generate IUPAC name from Molecule
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_networkx(self)</code>	Generate a NetworkX undirected graph from the Molecule.
<code>to_openeye(self[, aromaticity_model])</code>	Create an OpenEye molecule
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_rdkit(self[, aromaticity_model])</code>	Create an RDKit molecule
<code>to_smiles(self[, toolkit_registry])</code>	Return a canonical isomeric SMILES representation of the current molecule
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_topology(self)</code>	Return an openforcefield Topology representation containing one copy of this molecule
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

`__init__(self, other=None, file_format=None, toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>, allow_undefined_stereo=False)`  
 Create a new FrozenMolecule object

#### Parameters

**other** [optional, default=None] If specified, attempt to construct a copy of the Molecule from the specified object. This can be any one of the following:

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

**file\_format** [str, optional, default=None] If providing a file-like object, you must specify the format of the data. If providing a file, the file format will attempt to be guessed from the suffix.

**toolkit\_registry** [a `ToolkitRegistry` or `ToolkitWrapper` object, optional, default=GLOBAL\_TOOLKIT\_REGISTRY] `ToolkitRegistry` or `ToolkitWrapper` to use for I/O operations

**allow\_undefined\_stereo** [bool, default=False] If loaded from a file and `False`, raises an exception if undefined stereochemistry is detected during the molecule's construction.

#### Examples

Create an empty molecule:

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

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

```
>>> from openforcefield.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)
```

Create a molecule from a serialized molecule object:

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

## Methods

<code>__init__(self[, other, file_format, ...])</code>	Create a new FrozenMolecule object
<code>chemical_environment_matches(self, query[, ...])</code>	Retrieve all matches for a given chemical environment query.
<code>compute_partial_charges(self[, toolkit_registry])</code>	<b>Warning! Not Implemented!</b> Calculate partial atomic charges for this molecule using an underlying toolkit
<code>compute_partial_charges_am1bcc(self[, ...])</code>	Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit
<code>compute_wiberg_bond_orders(self[, ...])</code>	Calculate wiberg bond orders for this molecule using an underlying toolkit

Continued on next page

Table 3 – continued from previous page

<code>from_bson(serialized)</code>		Instantiate an object from a BSON serialized representation.
<code>from_dict(molecule_dict)</code>		Create a new Molecule from a dictionary representation
<code>from_file(file_path[, file_format, ...])</code>		Create one or more molecules from a file
<code>from_iupac(iupac_name, \**kwargs)</code>		Generate a molecule from IUPAC or common name
<code>from_json(serialized)</code>		Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>		Instantiate an object from a MessagePack serialized representation.
<code>from_openeye(oemol[, low_undefined_stereo])</code>	al-	Create a Molecule from an OpenEye molecule.
<code>from_pickle(serialized)</code>		Instantiate an object from a pickle serialized representation.
<code>from_rdkit(rdmol[, allow_undefined_stereo])</code>		Create a Molecule from an RDKit molecule.
<code>from_smiles(smiles[, ...])</code>		Construct a Molecule from a SMILES representation
<code>from_toml(serialized)</code>		Instantiate an object from a TOML serialized representation.
<code>from_topology(topology)</code>		Return a Molecule representation of an openforcefield Topology containing a single Molecule object.
<code>from_xml(serialized)</code>		Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>		Instantiate from a YAML serialized representation.
<code>generate_conformers(self[, ...])</code>		Generate conformers for this molecule using an underlying toolkit
<code>get_bond_between(self, i, j)</code>		Returns the bond between two atoms
<code>get_fractional_bond_orders(self[, method, ...])</code>		Get fractional bond orders.
<code>is_isomorphic(self, other[, ...])</code>		Determines whether the molecules are isomorphic by comparing their graphs.
<code>to_bson(self)</code>		Return a BSON serialized representation.
<code>to_dict(self)</code>		Return a dictionary representation of the molecule.
<code>to_file(self, file_path, file_format[, ...])</code>		Write the current molecule to a file or file-like object
<code>to_iupac(self)</code>		Generate IUPAC name from Molecule
<code>to_json(self[, indent])</code>		Return a JSON serialized representation.
<code>to_messagepack(self)</code>		Return a MessagePack representation.
<code>to_networkx(self)</code>		Generate a NetworkX undirected graph from the Molecule.
<code>to_openeye(self[, aromaticity_model])</code>		Create an OpenEye molecule
<code>to_pickle(self)</code>		Return a pickle serialized representation.
<code>to_rdkit(self[, aromaticity_model])</code>		Create an RDKit molecule
<code>to_smiles(self[, toolkit_registry])</code>		Return a canonical isomeric SMILES representation of the current molecule
<code>to_toml(self)</code>		Return a TOML serialized representation.

Continued on next page



Table 3 – continued from previous page

<code>to_topology(self)</code>	Return an openforcefield Topology representation containing one copy of this molecule
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

**Attributes**

<code>angles</code>	Get an iterator over all i-j-k angles.
<code>atoms</code>	Iterate over all Atom objects.
<code>bonds</code>	Iterate over all Bond objects.
<code>conformers</code>	Returns the list of conformers for this molecule.
<code>impropers</code>	Iterate over all proper torsions in the molecule
<code>n_angles</code>	int: number of angles in the Molecule.
<code>n_atoms</code>	The number of Atom objects.
<code>n_bonds</code>	The number of Bond objects.
<code>n_conformers</code>	Returns the number of conformers for this molecule.
<code>n_impropers</code>	int: number of improper torsions in the Molecule.
<code>n_particles</code>	The number of Particle objects, which corresponds to how many positions must be used.
<code>n_propers</code>	int: number of proper torsions in the Molecule.
<code>n_virtual_sites</code>	The number of VirtualSite objects.
<code>name</code>	The name (or title) of the molecule
<code>partial_charges</code>	Returns the partial charges (if present) on the molecule.
<code>particles</code>	Iterate over all Particle objects.
<code>propers</code>	Iterate over all proper torsions in the molecule
<code>properties</code>	The properties dictionary of the molecule
<code>torsions</code>	Get an iterator over all i-j-k-l torsions.
<code>total_charge</code>	Return the total charge on the molecule
<code>virtual_sites</code>	Iterate over all VirtualSite objects.

**to\_dict(self)**

Return a dictionary representation of the molecule.

**Returns**

**molecule\_dict** [OrderedDict] A dictionary representation of the molecule.

**classmethod from\_dict(molecule\_dict)**

Create a new Molecule from a dictionary representation

**Parameters**

**molecule\_dict** [OrderedDict] A dictionary representation of the molecule.

**Returns**

**molecule** [Molecule] A Molecule created from the dictionary representation

**to\_smiles(self, toolkit\_registry=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)**

Return a canonical isomeric SMILES representation of the current molecule

---

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

---

#### Parameters

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] ToolkitRegistry or ToolkitWrapper to use for SMILES conversion

#### Returns

**smiles** [str] Canonical isomeric explicit-hydrogen SMILES

#### Examples

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

**static from\_smiles**(smiles, hydrogens\_are\_explicit=False, toolkit\_registry=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>, allow\_undefined\_stereo=False)

Construct a Molecule from a SMILES representation

#### Parameters

**smiles** [str] The SMILES representation of the molecule.

**hydrogens\_are\_explicit** [bool, default = False] If False, the cheminformatics toolkit will perform hydrogen addition

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion

**allow\_undefined\_stereo** [bool, default=False] Whether to accept SMILES with undefined stereochemistry. If False, an exception will be raised if a SMILES with undefined stereochemistry is passed into this function.

#### Returns

**molecule** [openforcefield.topology.Molecule]

#### Examples

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

**is\_isomorphic**(self, other, compare\_atom\_stereochemistry=True, compare\_bond\_stereochemistry=True)

Determines whether the molecules are isomorphic by comparing their graphs.

#### Parameters

**other** [an openforcefield.topology.molecule.FrozenMolecule] The molecule to test for isomorphism.

**compare\_atom\_stereochemistry** [bool, optional] If False, atoms' stereochemistry is ignored for the purpose of determining equality. Default is True.

**compare\_bond\_stereochemistry** [bool, optional] If False, bonds' stereochemistry is ignored for the purpose of determining equality. Default is True.

#### Returns

**molecules\_are\_isomorphic** [bool]

**generate\_conformers**(*self*, *toolkit\_registry*=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>, *n\_conformers*=10, *clear\_existing*=True)  
Generate conformers for this molecule using an underlying toolkit

#### Parameters

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion

**n\_conformers** [int, default=1] The maximum number of conformers to produce

**clear\_existing** [bool, default=True] Whether to overwrite existing conformers for the molecule

#### Raises

**InvalidToolkitError** If an invalid object is passed as the *toolkit\_registry* parameter

#### Examples

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

**compute\_partial\_charges\_am1bcc**(*self*, *toolkit\_registry*=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)

Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit

#### Parameters

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] ToolkitRegistry or ToolkitWrapper to use for the calculation

#### Raises

**InvalidToolkitError** If an invalid object is passed as the *toolkit\_registry* parameter

#### Examples

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

**compute\_partial\_charges**(*self*, *toolkit\_registry*=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)

**Warning! Not Implemented!** Calculate partial atomic charges for this molecule using an underlying toolkit

#### Parameters

**quantum\_chemical\_method** [string, default='AM1-BCC'] The quantum chemical method to use for partial charge calculation.

**partial\_charge\_method** [string, default='None'] The partial charge calculation method to use for partial charge calculation.

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion

#### Raises

**InvalidToolkitError** If an invalid object is passed as the toolkit\_registry parameter

#### Examples

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

**compute\_wiberg\_bond\_orders**(self, charge\_model=None, toolkit\_registry=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)

Calculate wiberg bond orders for this molecule using an underlying toolkit

#### Parameters

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion

**charge\_model** [string, optional] The charge model to use for partial charge calculation

#### Raises

**InvalidToolkitError** If an invalid object is passed as the toolkit\_registry parameter

#### Examples

```
>>> molecule = Molecule.from_smiles('CCCCC')
>>> molecule.generate_conformers()
>>> molecule.compute_wiberg_bond_orders()
```

**to\_networkx**(self)

Generate a NetworkX undirected graph from the Molecule.

Nodes are Atoms labeled with particle indices and atomic elements (via the element node attribute). Edges denote chemical bonds between Atoms. Virtual sites are not included, since they lack a concept of chemical connectivity.

#### Returns

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

#### Examples

Retrieve the bond graph for imatinib (OpenEye toolkit required)

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

**property partial\_charges**

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

**Returns**

**partial\_charges** [a simtk.unit.Quantity - wrapped numpy array [1 x n\_atoms] or None] The partial charges on this Molecule's atoms. Returns None if no charges have been specified.

**property n\_particles**

The number of Particle objects, which corresponds to how many positions must be used.

**property n\_atoms**

The number of Atom objects.

**property n\_virtual\_sites**

The number of VirtualSite objects.

**property n\_bonds**

The number of Bond objects.

**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 improper torsions in the Molecule.

**property particles**

Iterate over all Particle objects.

**property atoms**

Iterate over all Atom objects.

**property conformers**

Returns the list of conformers for this molecule. This returns a list of simtk.unit.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**

Returns the number of conformers for this molecule.

**property virtual\_sites**

Iterate over all VirtualSite objects.

**property bonds**

Iterate over all Bond objects.

**property angles**

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

**property torsions**

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

**Returns**

**torsions** [iterable of 4-Atom tuples]

**property** `probers`

Iterate over all proper torsions in the molecule

**property** `improbers`

Iterate over all proper torsions in the molecule

**property** `total_charge`

Return the total charge on the molecule

**property** `name`

The name (or title) of the molecule

**property** `properties`

The properties dictionary of the molecule

**chemical\_environment\_matches**(*self*, *query*, *toolkit\_registry*=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)

Retrieve all matches for a given chemical environment query.

**Parameters**

**query** [str or ChemicalEnvironment] SMARTS string (with one or more tagged atoms) or ChemicalEnvironment query. Query will internally be resolved to SMIRKS using `query.asSMIRKS()` if it has an `.asSMIRKS` method.

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL\_TOOLKIT\_REGISTRY] ToolkitRegistry or ToolkitWrapper to use for chemical environment matches

**Returns**

**matches** [list of atom index tuples] A list of tuples, containing the indices of the matching atoms.

**Examples**

Retrieve all the carbon-carbon bond matches in a molecule

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

**classmethod** `from_iupac`(*iupac\_name*, *\*\*kwargs*)

Generate a molecule from IUPAC or common name

**Parameters**

**iupac\_name** [str] IUPAC name of molecule to be generated

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if molecule contains undefined stereochemistry.

**Returns**

**molecule** [Molecule] The resulting molecule with position

---

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

---

## Examples

Create a molecule from a common 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')
```

Create a molecule from a common name

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

**to\_iupac**(*self*)

Generate IUPAC name from Molecule

### Returns

**iupac\_name** [str] IUPAC name of the molecule

---

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

---

## Examples

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

**static from\_topology**(*topology*)

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

### Parameters

**topology** [openforcefield.topology.Topology] The [Topology](#) object containing a single [Molecule](#) object. Note that OpenMM and MDTraj Topology objects are not supported.

### Returns

**molecule** [openforcefield.topology.Molecule] The Molecule object in the topology

### Raises

**ValueError** If the topology does not contain exactly one molecule.

## Examples

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

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

**to\_topology**(*self*)

Return an openforcefield Topology representation containing one copy of this molecule

### Returns

**topology** [openforcefield.topology.Topology] A Topology representation of this molecule

### Examples

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

**static from\_file**(file\_path, file\_format=None, toolkit\_registry=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>, allow\_undefined\_stereo=False)

Create one or more molecules from a file

#### Parameters

**file\_path** [str or file-like object] The path to the file or file-like object to stream one or more molecules from.

**file\_format** [str, optional, default=None] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit\_file\_read\_formats for your loaded toolkits for details.

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper,]

**optional, default=GLOBAL\_TOOLKIT\_REGISTRY** ToolkitRegistry or ToolkitWrapper to use for file loading. If a Toolkit is passed, only the highest-precedence toolkit is used

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

#### Returns

**molecules** [Molecule or list of Molecules] If there is a single molecule in the file, a Molecule is returned; otherwise, a list of Molecule objects is returned.

### Examples

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

**to\_file**(self, file\_path, file\_format, toolkit\_registry=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)

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

#### Parameters

**file\_path** [str or file-like object] A file-like object or the path to the file to be written.

**file\_format** [str] Format specifier, one of ['MOL2', 'MOL2H', 'SDF', 'PDB', 'SMI', 'CAN', 'TDT'] Note that not all toolkits support all formats

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper,]

**optional, default=GLOBAL\_TOOLKIT\_REGISTRY** ToolkitRegistry or ToolkitWrapper to use for file writing. If a Toolkit is passed, only the highest-precedence toolkit is used



**Raises**

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

**Examples**

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

**static** `from_rdkit(rdmol, allow_undefined_stereo=False)`

Create a Molecule from an RDKit molecule.

Requires the RDKit to be installed.

**Parameters**

**rdmol** [`rdkit.RDMol`] An RDKit molecule

**allow\_undefined\_stereo** [`bool`, default=`False`] If false, raises an exception if oemol contains undefined stereochemistry.

**Returns**

**molecule** [`openforcefield.Molecule`] An openforcefield molecule

**Examples**

Create a molecule from an RDKit molecule

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

**to\_rdkit**(*self*, `aromaticity_model='OEArModel_MDL'`)

Create an RDKit molecule

Requires the RDKit to be installed.

**Parameters**

**aromaticity\_model** [`str`, optional, default=`DEFAULT_AROMATICITY_MODEL`] The aromaticity model to use

**Returns**

**rdmol** [`rdkit.RDMol`] An RDKit molecule

**Examples**

Convert a molecule to RDKit

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

**static from\_openeye**(*oemol*, *allow\_undefined\_stereo=False*)

Create a Molecule from an OpenEye molecule.

Requires the OpenEye toolkit to be installed.

#### Parameters

**oemol** [openeye.oechem.OEMol] An OpenEye molecule

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

#### Returns

**molecule** [openforcefield.topology.Molecule] An openforcefield molecule

### Examples

Create a Molecule from an OpenEye OEMol

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

**to\_openeye**(*self*, *aromaticity\_model='OEModel\_MDL'*)

Create an OpenEye molecule

Requires the OpenEye toolkit to be installed.

#### Parameters

**aromaticity\_model** [str, optional, default=DEFAULT\_AROMATICITY\_MODEL] The aromaticity model to use

#### Returns

**oemol** [openeye.oechem.OEMol] An OpenEye molecule

### Examples

Create an OpenEye molecule from a Molecule

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

**get\_fractional\_bond\_orders**(*self*, *method='Wiberg'*, *toolkit\_registry=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>*)

Get fractional bond orders.

**method** [str, optional, default='Wiberg'] The name of the charge method to use. Options are: \*  
'Wiberg': Wiberg bond order

**toolkit\_registry** [openforcefield.utils.toolkits.ToolkitRegistry] The toolkit registry to use for molecule operations

## Examples

Get fractional Wiberg bond orders

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> molecule.generate_conformers()
>>> fractional_bond_orders = molecule.get_fractional_bond_orders(method='Wiberg')
```

**classmethod** `from_bson(serialized)`

Instantiate an object from a BSON serialized representation.

Specification: <http://bsonspec.org/>

### Parameters

**serialized** [bytes] A BSON serialized representation of the object

### Returns

**instance** [cls] An instantiated object

**classmethod** `from_json(serialized)`

Instantiate an object from a JSON serialized representation.

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

### Parameters

**serialized** [str] A JSON serialized representation of the object

### Returns

**instance** [cls] An instantiated object

**classmethod** `from_messagepack(serialized)`

Instantiate an object from a MessagePack serialized representation.

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

### Parameters

**serialized** [bytes] A MessagePack-encoded bytes serialized representation

### Returns

**instance** [cls] Instantiated object.

**classmethod** `from_pickle(serialized)`

Instantiate an object from a pickle serialized representation.

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

### Parameters

**serialized** [str] A pickled representation of the object

### Returns

**instance** [cls] An instantiated object

**classmethod** `from_toml(serialized)`

Instantiate an object from a TOML serialized representation.

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

**Parameters**

**serialized** [str] A TOML serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod** `from_xml(serialized)`

Instantiate an object from an XML serialized representation.

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

**Parameters**

**serialized** [bytes] An XML serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod** `from_yaml(serialized)`

Instantiate from a YAML serialized representation.

Specification: <http://yaml.org/>

**Parameters**

**serialized** [str] A YAML serialized representation of the object

**Returns**

**instance** [cls] Instantiated object

**to\_bson**(*self*)

Return a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Returns**

**serialized** [bytes] A BSON serialized representation of the object

**to\_json**(*self*, *indent=None*)

Return a JSON serialized representation.

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

**Parameters**

**indent** [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

**Returns**

**serialized** [str] A JSON serialized representation of the object

**to\_messagepack**(*self*)

Return a MessagePack representation.

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

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation of the object

**to\_pickle**(*self*)

Return a pickle serialized representation.

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

#### Returns

**serialized** [str] A pickled representation of the object

**to\_toml**(*self*)

Return a TOML serialized representation.

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

#### Returns

**serialized** [str] A TOML serialized representation of the object

**to\_xml**(*self*, *indent*=2)

Return an XML representation.

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

#### Parameters

**indent** [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

#### Returns

**serialized** [bytes] A MessagePack-encoded bytes serialized representation.

**to\_yaml**(*self*)

Return a YAML serialized representation.

Specification: <http://yaml.org/>

#### Returns

**serialized** [str] A YAML serialized representation of the object

**get\_bond\_between**(*self*, *i*, *j*)

Returns the bond between two atoms

#### Parameters

**i, j** [int or Atom] Atoms or atom indices to check

#### Returns

**bond** [Bond] The bond between *i* and *j*.

## openforcefield.topology.Molecule

**class** openforcefield.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 openforcefield.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.

## Attributes

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

**atoms** Iterate over all Atom objects.

**bonds** Iterate over all Bond objects.

**conformers** Returns the list of conformers for this molecule.

**impropers** Iterate over all proper torsions in the molecule

**n\_angles** int: number of angles in the Molecule.

**n\_atoms** The number of Atom objects.

**n\_bonds** The number of Bond objects.

**n\_conformers** Returns the number of conformers for this molecule.

**n\_impropers** int: number of improper torsions in the Molecule.

**n\_particles** The number of Particle objects, which corresponds to how many positions must be used.

**n\_propers** int: number of proper torsions in the Molecule.

**n\_virtual\_sites** The number of VirtualSite objects.

**name** The name (or title) of the molecule

**partial\_charges** Returns the partial charges (if present) on the molecule.

**particles** Iterate over all Particle objects.

**propers** Iterate over all proper torsions in the molecule

**properties** The properties dictionary of the molecule

**torsions** Get an iterator over all i-j-k-l torsions.

**total\_charge** Return the total charge on the molecule

**virtual\_sites** Iterate over all VirtualSite objects.

## Methods

<code>add_atom(self, atomic_number, formal_charge, ...)</code>	Add an atom
<code>add_bond(self, atom1, atom2, bond_order, ...)</code>	Add a bond between two specified atom indices
<code>add_bond_charge_virtual_site(self, atoms, ...)</code>	Create a bond charge-type virtual site, in which the location of the charge is specified by the position of two atoms.
<code>add_conformer(self, coordinates)</code>	Add a conformation of the molecule
<code>add_divalent_lone_pair_virtual_site(self, ...)</code>	Create a divalent lone pair-type virtual site, in which the location of the charge is specified by the position of three atoms.
<code>add_monovalent_lone_pair_virtual_site(self, ...)</code>	Create a bond charge-type virtual site, in which the location of the charge is specified by the position of three atoms.
<code>add_trivalent_lone_pair_virtual_site(self, ...)</code>	Create a trivalent lone pair-type virtual site, in which the location of the charge is specified by the position of four atoms.
<code>chemical_environment_matches(self, query[, ...])</code>	Retrieve all matches for a given chemical environment query.
<code>compute_partial_charges(self[, toolkit_registry])</code>	<b>Warning! Not Implemented!</b> Calculate partial atomic charges for this molecule using an underlying toolkit
<code>compute_partial_charges_am1bcc(self[, ...])</code>	Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit
<code>compute_wiberg_bond_orders(self[, ...])</code>	Calculate wiberg bond orders for this molecule using an underlying toolkit
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(molecule_dict)</code>	Create a new Molecule from a dictionary representation
<code>from_file(file_path[, file_format, ...])</code>	Create one or more molecules from a file
<code>from_iupac(iupac_name, \**kwargs)</code>	Generate a molecule from IUPAC or common name
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.

Continued on next page

Table 5 – continued from previous page

<code>from_messagepack(serialized)</code>		Instantiate an object from a MessagePack serialized representation.
<code>from_openeye(oemol[, low_undefined_stereo])</code>	al-	Create a Molecule from an OpenEye molecule.
<code>from_pickle(serialized)</code>		Instantiate an object from a pickle serialized representation.
<code>from_rdkit(rdmol[, allow_undefined_stereo])</code>		Create a Molecule from an RDKit molecule.
<code>from_smiles(smiles[, ...])</code>		Construct a Molecule from a SMILES representation
<code>from_toml(serialized)</code>		Instantiate an object from a TOML serialized representation.
<code>from_topology(topology)</code>		Return a Molecule representation of an openforcefield Topology containing a single Molecule object.
<code>from_xml(serialized)</code>		Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>		Instantiate from a YAML serialized representation.
<code>generate_conformers(self[, ...])</code>		Generate conformers for this molecule using an underlying toolkit
<code>get_bond_between(self, i, j)</code>		Returns the bond between two atoms
<code>get_fractional_bond_orders(self[, ...])</code>	method,	Get fractional bond orders.
<code>is_isomorphic(self, other[, ...])</code>		Determines whether the molecules are isomorphic by comparing their graphs.
<code>to_bson(self)</code>		Return a BSON serialized representation.
<code>to_dict(self)</code>		Return a dictionary representation of the molecule.
<code>to_file(self, file_path, file_format[, ...])</code>		Write the current molecule to a file or file-like object
<code>to_iupac(self)</code>		Generate IUPAC name from Molecule
<code>to_json(self[, indent])</code>		Return a JSON serialized representation.
<code>to_messagepack(self)</code>		Return a MessagePack representation.
<code>to_networkx(self)</code>		Generate a NetworkX undirected graph from the Molecule.
<code>to_openeye(self[, aromaticity_model])</code>		Create an OpenEye molecule
<code>to_pickle(self)</code>		Return a pickle serialized representation.
<code>to_rdkit(self[, aromaticity_model])</code>		Create an RDKit molecule
<code>to_smiles(self[, toolkit_registry])</code>		Return a canonical isomeric SMILES representation of the current molecule
<code>to_toml(self)</code>		Return a TOML serialized representation.
<code>to_topology(self)</code>		Return an openforcefield Topology representation containing one copy of this molecule
<code>to_xml(self[, indent])</code>		Return an XML representation.
<code>to_yaml(self)</code>		Return a YAML serialized representation.

`__init__(self, *args, **kwargs)`  
Create a new Molecule object

#### Parameters

**other** [optional, default=None] If specified, attempt to construct a copy of the Molecule from the specified object. This can be any one of the following:



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

## Examples

Create an empty molecule:

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

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

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

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

Create a molecule from another molecule:

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

Convert to OpenEye OEMol object

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

Create a molecule from an OpenEye molecule:

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

Convert to RDKit Mol object

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

Create a molecule from an RDKit molecule:

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

Create a molecule from a serialized molecule object:

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

## Methods

<code>__init__(self, \*args, \*\*kwargs)</code>	Create a new Molecule object
<code>add_atom(self, atomic_number, formal_charge, ...)</code>	Add an atom
<code>add_bond(self, atom1, atom2, bond_order, ...)</code>	Add a bond between two specified atom indices
<code>add_bond_charge_virtual_site(self, atoms, ...)</code>	Create a bond charge-type virtual site, in which the location of the charge is specified by the position of two atoms.
<code>add_conformer(self, coordinates)</code>	Add a conformation of the molecule
<code>add_divalent_lone_pair_virtual_site(self, ...)</code>	Create a divalent lone pair-type virtual site, in which the location of the charge is specified by the position of three atoms.
<code>add_monovalent_lone_pair_virtual_site(self, ...)</code>	Create a bond charge-type virtual site, in which the location of the charge is specified by the position of three atoms.
<code>add_trivalent_lone_pair_virtual_site(self, ...)</code>	Create a trivalent lone pair-type virtual site, in which the location of the charge is specified by the position of four atoms.
<code>chemical_environment_matches(self, query, ...)</code>	Retrieve all matches for a given chemical environment query.
<code>compute_partial_charges(self, toolkit_registry)</code>	<b>Warning! Not Implemented!</b> Calculate partial atomic charges for this molecule using an underlying toolkit
<code>compute_partial_charges_am1bcc(self, ...)</code>	Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit
<code>compute_wiberg_bond_orders(self, ...)</code>	Calculate wiberg bond orders for this molecule using an underlying toolkit
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(molecule_dict)</code>	Create a new Molecule from a dictionary representation
<code>from_file(file_path[, file_format, ...])</code>	Create one or more molecules from a file
<code>from_iupac(iupac_name, \*\*kwargs)</code>	Generate a molecule from IUPAC or common name
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_openeye(oemol[, allow_undefined_stereo])</code>	Create a Molecule from an OpenEye molecule.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_rdkit(rdmol[, allow_undefined_stereo])</code>	Create a Molecule from an RDKit molecule.
<code>from_smiles(smiles[, ...])</code>	Construct a Molecule from a SMILES representation
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_topology(topology)</code>	Return a Molecule representation of an openforcefield Topology containing a single Molecule object.

Continued on next page

Table 6 – continued from previous page

<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>generate_conformers(self[, ...])</code>	Generate conformers for this molecule using an underlying toolkit
<code>get_bond_between(self, i, j)</code>	Returns the bond between two atoms
<code>get_fractional_bond_orders(self[, method, ...])</code>	Get fractional bond orders.
<code>is_isomorphic(self, other[, ...])</code>	Determines whether the molecules are isomorphic by comparing their graphs.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Return a dictionary representation of the molecule.
<code>to_file(self, file_path, file_format[, ...])</code>	Write the current molecule to a file or file-like object
<code>to_iupac(self)</code>	Generate IUPAC name from Molecule
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_networkx(self)</code>	Generate a NetworkX undirected graph from the Molecule.
<code>to_openeye(self[, aromaticity_model])</code>	Create an OpenEye molecule
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_rdkit(self[, aromaticity_model])</code>	Create an RDKit molecule
<code>to_smiles(self[, toolkit_registry])</code>	Return a canonical isomeric SMILES representation of the current molecule
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_topology(self)</code>	Return an openforcefield Topology representation containing one copy of this molecule
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

**Attributes**

<code>angles</code>	Get an iterator over all i-j-k angles.
<code>atoms</code>	Iterate over all Atom objects.
<code>bonds</code>	Iterate over all Bond objects.
<code>conformers</code>	Returns the list of conformers for this molecule.
<code>impropers</code>	Iterate over all proper torsions in the molecule
<code>n_angles</code>	int: number of angles in the Molecule.
<code>n_atoms</code>	The number of Atom objects.
<code>n_bonds</code>	The number of Bond objects.
<code>n_conformers</code>	Returns the number of conformers for this molecule.
<code>n_impropers</code>	int: number of improper torsions in the Molecule.
<code>n_particles</code>	The number of Particle objects, which corresponds to how many positions must be used.
<code>n_propers</code>	int: number of proper torsions in the Molecule.
<code>n_virtual_sites</code>	The number of VirtualSite objects.

Continued on next page

Table 7 – continued from previous page

<code>name</code>	The name (or title) of the molecule
<code>partial_charges</code>	Returns the partial charges (if present) on the molecule.
<code>particles</code>	Iterate over all Particle objects.
<code>probers</code>	Iterate over all proper torsions in the molecule
<code>properties</code>	The properties dictionary of the molecule
<code>torsions</code>	Get an iterator over all i-j-k-l torsions.
<code>total_charge</code>	Return the total charge on the molecule
<code>virtual_sites</code>	Iterate over all VirtualSite objects.

**property angles**

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

**property atoms**

Iterate over all Atom objects.

**property bonds**

Iterate over all Bond objects.

**chemical\_environment\_matches**(*self*, *query*, *toolkit\_registry*=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)

Retrieve all matches for a given chemical environment query.

**Parameters**

**query** [str or ChemicalEnvironment] SMARTS string (with one or more tagged atoms) or ChemicalEnvironment query. Query will internally be resolved to SMIRKS using `query.asSMIRKS()` if it has an `.asSMIRKS` method.

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL\_TOOLKIT\_REGISTRY] ToolkitRegistry or ToolkitWrapper to use for chemical environment matches

**Returns**

**matches** [list of atom index tuples] A list of tuples, containing the indices of the matching atoms.

**Examples**

Retrieve all the carbon-carbon bond matches in a molecule

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

**compute\_partial\_charges**(*self*, *toolkit\_registry*=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)

**Warning! Not Implemented!** Calculate partial atomic charges for this molecule using an underlying toolkit

**Parameters**

**quantum\_chemical\_method** [string, default='AM1-BCC'] The quantum chemical method to use for partial charge calculation.

**partial\_charge\_method** [string, default='None'] The partial charge calculation method to use for partial charge calculation.

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion

#### Raises

**InvalidToolkitError** If an invalid object is passed as the toolkit\_registry parameter

#### Examples

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

**compute\_partial\_charges\_am1bcc**(self, toolkit\_registry=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)

Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit

#### Parameters

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] ToolkitRegistry or ToolkitWrapper to use for the calculation

#### Raises

**InvalidToolkitError** If an invalid object is passed as the toolkit\_registry parameter

#### Examples

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

**compute\_wiberg\_bond\_orders**(self, charge\_model=None, toolkit\_registry=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)

Calculate wiberg bond orders for this molecule using an underlying toolkit

#### Parameters

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion

**charge\_model** [string, optional] The charge model to use for partial charge calculation

#### Raises

**InvalidToolkitError** If an invalid object is passed as the toolkit\_registry parameter

#### Examples

```
>>> molecule = Molecule.from_smiles('CCCCC')
>>> molecule.generate_conformers()
>>> molecule.compute_wiberg_bond_orders()
```

**property conformers**

Returns the list of conformers for this molecule. This returns a list of `simtk.unit.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`.

**classmethod from\_bson(serialized)**

Instantiate an object from a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Parameters**

**serialized** [bytes] A BSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_dict(molecule\_dict)**

Create a new `Molecule` from a dictionary representation

**Parameters**

**molecule\_dict** [OrderedDict] A dictionary representation of the molecule.

**Returns**

**molecule** [`Molecule`] A `Molecule` created from the dictionary representation

**static from\_file(file\_path, file\_format=None, toolkit\_registry=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>, allow\_undefined\_stereo=False)**

Create one or more molecules from a file

**Parameters**

**file\_path** [str or file-like object] The path to the file or file-like object to stream one or more molecules from.

**file\_format** [str, optional, default=None] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check `ToolkitWrapper.toolkit_file_read_formats` for your loaded toolkits for details.

**toolkit\_registry** [`openforcefield.utils.toolkits.ToolRegistry` or `openforcefield.utils.toolkits.ToolkitWrapper`,]

**optional, default=GLOBAL\_TOOLKIT\_REGISTRY** `ToolkitRegistry` or `ToolkitWrapper` to use for file loading. If a `Toolkit` is passed, only the highest-precedence toolkit is used

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

**Returns**

**molecules** [`Molecule` or list of `Molecules`] If there is a single molecule in the file, a `Molecule` is returned; otherwise, a list of `Molecule` objects is returned.

**Examples**

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

**classmethod** `from_iupac(iupac_name, **kwargs)`  
 Generate a molecule from IUPAC or common name

**Parameters**

**iupac\_name** [str] IUPAC name of molecule to be generated

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if molecule contains undefined stereochemistry.

**Returns**

**molecule** [Molecule] The resulting molecule with position

---

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

---

**Examples**

Create a molecule from a common 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')
```

Create a molecule from a common name

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

**classmethod** `from_json(serialized)`  
 Instantiate an object from a JSON serialized representation.

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

**Parameters**

**serialized** [str] A JSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod** `from_messagepack(serialized)`  
 Instantiate an object from a MessagePack serialized representation.

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

**Parameters**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation

**Returns**

**instance** [cls] Instantiated object.

**static** `from_openeye(oemol, allow_undefined_stereo=False)`  
 Create a Molecule from an OpenEye molecule.

Requires the OpenEye toolkit to be installed.

**Parameters**

**oemol** [openeye.oechem.OEMol] An OpenEye molecule

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

#### Returns

**molecule** [openforcefield.topology.Molecule] An openforcefield molecule

### Examples

Create a Molecule from an OpenEye OEMol

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

**classmethod from\_pickle**(*serialized*)

Instantiate an object from a pickle serialized representation.

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

#### Parameters

**serialized** [str] A pickled representation of the object

#### Returns

**instance** [cls] An instantiated object

**static from\_rdkit**(*rdmol*, *allow\_undefined\_stereo=False*)

Create a Molecule from an RDKit molecule.

Requires the RDKit to be installed.

#### Parameters

**rdmol** [rkit.RDMol] An RDKit molecule

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

#### Returns

**molecule** [openforcefield.Molecule] An openforcefield molecule

### Examples

Create a molecule from an RDKit molecule

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



**static from\_smiles**(*smiles*, *hydrogens\_are\_explicit*=False, *toolkit\_registry*=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>, *allow\_undefined\_stereo*=False)

Construct a Molecule from a SMILES representation

#### Parameters

**smiles** [str] The SMILES representation of the molecule.

**hydrogens\_are\_explicit** [bool, default = False] If False, the cheminformatics toolkit will perform hydrogen addition

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion

**allow\_undefined\_stereo** [bool, default=False] Whether to accept SMILES with undefined stereochemistry. If False, an exception will be raised if a SMILES with undefined stereochemistry is passed into this function.

#### Returns

**molecule** [openforcefield.topology.Molecule]

#### Examples

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

**classmethod from\_toml**(*serialized*)

Instantiate an object from a TOML serialized representation.

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

#### Parameters

**serialized** [str] A TOML serialized representation of the object

#### Returns

**instance** [cls] An instantiated object

**static from\_topology**(*topology*)

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

#### Parameters

**topology** [openforcefield.topology.Topology] The [Topology](#) object containing a single [Molecule](#) object. Note that OpenMM and MDTraj Topology objects are not supported.

#### Returns

**molecule** [openforcefield.topology.Molecule] The Molecule object in the topology

#### Raises

**ValueError** If the topology does not contain exactly one molecule.

#### Examples

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

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

**classmethod** `from_xml(serialized)`

Instantiate an object from an XML serialized representation.

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

**Parameters**

**serialized** [bytes] An XML serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod** `from_yaml(serialized)`

Instantiate from a YAML serialized representation.

Specification: <http://yaml.org/>

**Parameters**

**serialized** [str] A YAML serialized representation of the object

**Returns**

**instance** [cls] Instantiated object

**generate\_conformers**(*self*, *toolkit\_registry*=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>, *n\_conformers*=10, *clear\_existing*=True)

Generate conformers for this molecule using an underlying toolkit

**Parameters**

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion

**n\_conformers** [int, default=1] The maximum number of conformers to produce

**clear\_existing** [bool, default=True] Whether to overwrite existing conformers for the molecule

**Raises**

**InvalidToolkitError** If an invalid object is passed as the *toolkit\_registry* parameter

**Examples**

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

**get\_bond\_between**(*self*, *i*, *j*)

Returns the bond between two atoms

**Parameters**

**i, j** [int or Atom] Atoms or atom indices to check

**Returns**

**bond** [Bond] The bond between *i* and *j*.

**get\_fractional\_bond\_orders**(*self*, *method*='Wiberg', *toolkit\_registry*=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)

Get fractional bond orders.

**method** [str, optional, default='Wiberg'] The name of the charge method to use. Options are: \*  
'Wiberg': Wiberg bond order

**toolkit\_registry** [openforcefield.utils.toolkits.ToolkitRegistry] The toolkit registry to use for molecule operations

## Examples

Get fractional Wiberg bond orders

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> molecule.generate_conformers()
>>> fractional_bond_orders = molecule.get_fractional_bond_orders(method='Wiberg')
```

## property impropers

Iterate over all proper torsions in the molecule

**is\_isomorphic**(*self*, *other*, *compare\_atom\_stereochemistry*=True, *compare\_bond\_stereochemistry*=True)

Determines whether the molecules are isomorphic by comparing their graphs.

### Parameters

**other** [an openforcefield.topology.molecule.FrozenMolecule] The molecule to test for isomorphism.

**compare\_atom\_stereochemistry** [bool, optional] If False, atoms' stereochemistry is ignored for the purpose of determining equality. Default is True.

**compare\_bond\_stereochemistry** [bool, optional] If False, bonds' stereochemistry is ignored for the purpose of determining equality. Default is True.

### Returns

**molecules\_are\_isomorphic** [bool]

## property n\_angles

int: number of angles in the Molecule.

## property n\_atoms

The number of Atom objects.

## property n\_bonds

The number of Bond objects.

## property n\_conformers

Returns the number of conformers for this molecule.

## property n\_improbers

int: number of improper torsions in the Molecule.

## property n\_particles

The number of Particle objects, which corresponds to how many positions must be used.

## property n\_probers

int: number of proper torsions in the Molecule.

**property n\_virtual\_sites**

The number of VirtualSite objects.

**property name**

The name (or title) of the molecule

**property partial\_charges**

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

**Returns**

**partial\_charges** [a simtk.unit.Quantity - wrapped numpy array [1 x n\_atoms] or None] The partial charges on this Molecule's atoms. Returns None if no charges have been specified.

**property particles**

Iterate over all Particle objects.

**property propers**

Iterate over all proper torsions in the molecule

**property properties**

The properties dictionary of the molecule

**to\_bson(self)**

Return a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Returns**

**serialized** [bytes] A BSON serialized representation of the object

**to\_dict(self)**

Return a dictionary representation of the molecule.

**Returns**

**molecule\_dict** [OrderedDict] A dictionary representation of the molecule.

**to\_file(self, file\_path, file\_format, toolkit\_registry=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)**

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

**Parameters**

**file\_path** [str or file-like object] A file-like object or the path to the file to be written.

**file\_format** [str] Format specifier, one of ['MOL2', 'MOL2H', 'SDF', 'PDB', 'SMI', 'CAN', 'TDT'] Note that not all toolkits support all formats

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper,]

**optional, default=GLOBAL\_TOOLKIT\_REGISTRY** ToolkitRegistry or ToolkitWrapper to use for file writing. If a Toolkit is passed, only the highest-precedence toolkit is used

**Raises**

**ValueError** If the requested file\_format is not supported by one of the installed chem-informatics 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_iupac(self)`

Generate IUPAC name from Molecule

#### Returns

**iupac\_name** [str] IUPAC name of the molecule

---

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

---

## Examples

```
>>> from openforcefield.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(self, indent=None)`

Return a JSON serialized representation.

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

#### Parameters

**indent** [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

#### Returns

**serialized** [str] A JSON serialized representation of the object

### `to_messagepack(self)`

Return a MessagePack representation.

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

#### Returns

**serialized** [bytes] A MessagePack-encoded bytes serialized representation of the object

### `to_networkx(self)`

Generate a NetworkX undirected graph from the Molecule.

Nodes are Atoms labeled with particle indices and atomic elements (via the `element` node attribute). Edges denote chemical bonds between Atoms. Virtual sites are not included, since they lack a concept of chemical connectivity.

#### Returns

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

## Examples

Retrieve the bond graph for imatinib (OpenEye toolkit required)

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

**to\_openeye**(*self*, aromaticity\_model='OEAroModel\_MDL')

Create an OpenEye molecule

Requires the OpenEye toolkit to be installed.

### Parameters

**aromaticity\_model** [str, optional, default=DEFAULT\_AROMATICITY\_MODEL] The aromaticity model to use

### Returns

**oemol** [openeye.oechem.OEMol] An OpenEye molecule

## Examples

Create an OpenEye molecule from a Molecule

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

**to\_pickle**(*self*)

Return a pickle serialized representation.

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

### Returns

**serialized** [str] A pickled representation of the object

**to\_rdkit**(*self*, aromaticity\_model='OEAroModel\_MDL')

Create an RDKit molecule

Requires the RDKit to be installed.

### Parameters

**aromaticity\_model** [str, optional, default=DEFAULT\_AROMATICITY\_MODEL] The aromaticity model to use

### Returns

**rdmol** [rdkit.RDMol] An RDKit molecule

## Examples

Convert a molecule to RDKit

```
>>> from openforcefield.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**(self, toolkit\_registry=<openforcefield.utils.toolkits.ToolkitRegistry object at 0x7f769344efd0>)  
Return a canonical isomeric SMILES representation of the current molecule

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

#### Parameters

**toolkit\_registry** [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] ToolkitRegistry or ToolkitWrapper to use for SMILES conversion

#### Returns

**smiles** [str] Canonical isomeric explicit-hydrogen SMILES

#### Examples

```
>>> from openforcefield.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**(self)  
Return a TOML serialized representation.

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

#### Returns

**serialized** [str] A TOML serialized representation of the object

**to\_topology**(self)  
Return an openforcefield Topology representation containing one copy of this molecule

#### Returns

**topology** [openforcefield.topology.Topology] A Topology representation of this molecule

#### Examples

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

**to\_xml**(self, indent=2)  
Return an XML representation.

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

#### Parameters

**indent** [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

#### Returns

**serialized** [bytes] A MessagePack-encoded bytes serialized representation.

**to\_yaml(self)**

Return a YAML serialized representation.

Specification: <http://yaml.org/>

#### Returns

**serialized** [str] A YAML serialized representation of the object

**property torsions**

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

#### Returns

**torsions** [iterable of 4-Atom tuples]

**property total\_charge**

Return the total charge on the molecule

**property virtual\_sites**

Iterate over all VirtualSite objects.

**add\_atom(self, atomic\_number, formal\_charge, is\_aromatic, stereochemistry=None, name=None)**

Add an atom

#### Parameters

**atomic\_number** [int] Atomic number of the atom

**formal\_charge** [int] Formal charge of the atom

**is\_aromatic** [bool] If True, atom is aromatic; if False, not aromatic

**stereochemistry** [str, optional, default=None] Either 'R' or 'S' for specified stereochemistry, or None if stereochemistry is irrelevant

**name** [str, optional, default=None] An optional name for the atom

#### Returns

**index** [int] The index of the atom in the molecule

## Examples

Define a methane molecule

```
>>> molecule = Molecule()
>>> molecule.name = 'methane'
>>> C = molecule.add_atom(6, 0, False)
>>> H1 = molecule.add_atom(1, 0, False)
>>> H2 = molecule.add_atom(1, 0, False)
>>> H3 = molecule.add_atom(1, 0, False)
>>> H4 = molecule.add_atom(1, 0, False)
>>> bond_idx = molecule.add_bond(C, H1, False, 1)
>>> bond_idx = molecule.add_bond(C, H2, False, 1)
>>> bond_idx = molecule.add_bond(C, H3, False, 1)
>>> bond_idx = molecule.add_bond(C, H4, False, 1)
```



**add\_bond\_charge\_virtual\_site**(*self*, *atoms*, *distance*, *charge\_increments*=None, *weights*=None, *epsilon*=None, *sigma*=None, *rmin\_half*=None, *name*=")

Create a bond charge-type virtual site, in which the location of the charge is specified by the position of two atoms. This supports placement of a virtual site S along a vector between two specified atoms, e.g. to allow for a sigma hole for halogens or similar contexts. With positive values of the distance, the virtual site lies outside the first indexed atom. Parameters ———  
*atoms* : list of openforcefield.topology.molecule.Atom objects or ints of shape [N]

The atoms defining the virtual site's position or their indices

*distance* : float

**weights** [list of floats of shape [N] or None, optional, default=None] *weights*[index] is the weight of particles[index] contributing to the position of the virtual site. Default is None

**charge\_increments** [list of floats of shape [N], optional, default=None] The amount of charge to remove from the VirtualSite's atoms and put in the VirtualSite. Indexing in this list should match the ordering in the atoms list. Default is None.

**epsilon** [float] Epsilon term for VdW properties of virtual site. Default is None.

**sigma** [float, default=None] Sigma term for VdW properties of virtual site. Default is None.

**rmin\_half** [float] Rmin\_half term for VdW properties of virtual site. Default is None.

**name** [string or None, default=""] The name of this virtual site. Default is "".

#### Returns

**index** [int] The index of the newly-added virtual site in the molecule

**add\_monovalent\_lone\_pair\_virtual\_site**(*self*, *atoms*, *distance*, *out\_of\_plane\_angle*, *in\_plane\_angle*, \*\*kwargs)

Create a bond charge-type virtual site, in which the location of the charge is specified by the position of three atoms.

TODO : Do "weights" have any meaning here?

#### Parameters

**atoms** [list of three openforcefield.topology.molecule.Atom objects or ints] The three atoms defining the virtual site's position or their molecule atom indices

**distance** [float]

**out\_of\_plane\_angle** [float]

**in\_plane\_angle** [float]

**epsilon** [float] Epsilon term for VdW properties of virtual site. Default is None.

**sigma** [float, default=None] Sigma term for VdW properties of virtual site. Default is None.

**rmin\_half** [float] Rmin\_half term for VdW properties of virtual site. Default is None.

**name** [string or None, default=""] The name of this virtual site. Default is "".

#### Returns

**index** [int] The index of the newly-added virtual site in the molecule

**add\_divalent\_lone\_pair\_virtual\_site**(*self*, *atoms*, *distance*, *out\_of\_plane\_angle*, *in\_plane\_angle*,  
\*\**kwargs*)

Create a divalent lone pair-type virtual site, in which the location of the charge is specified by the position of three atoms.

TODO : Do “weights” have any meaning here?

#### Parameters

**atoms** [list of 3 openforcefield.topology.molecule.Atom objects or ints] The three atoms defining the virtual site’s position or their molecule atom indices

**distance** [float]

**out\_of\_plane\_angle** [float]

**in\_plane\_angle** [float]

**epsilon** [float] Epsilon term for VdW properties of virtual site. Default is None.

**sigma** [float, default=None] Sigma term for VdW properties of virtual site. Default is None.

**rmin\_half** [float] Rmin\_half term for VdW properties of virtual site. Default is None.

**name** [string or None, default=”] The name of this virtual site. Default is ”.

#### Returns

**index** [int] The index of the newly-added virtual site in the molecule

**add\_trivalent\_lone\_pair\_virtual\_site**(*self*, *atoms*, *distance*, *out\_of\_plane\_angle*,  
*in\_plane\_angle*, \*\**kwargs*)

Create a trivalent lone pair-type virtual site, in which the location of the charge is specified by the position of four atoms.

TODO : Do “weights” have any meaning here?

#### Parameters

**atoms** [list of 4 openforcefield.topology.molecule.Atom objects or ints] The three atoms defining the virtual site’s position or their molecule atom indices

**distance** [float]

**out\_of\_plane\_angle** [float]

**in\_plane\_angle** [float]

**epsilon** [float] Epsilon term for VdW properties of virtual site. Default is None.

**sigma** [float, default=None] Sigma term for VdW properties of virtual site. Default is None.

**rmin\_half** [float] Rmin\_half term for VdW properties of virtual site. Default is None.

**name** [string or None, default=”] The name of this virtual site. Default is ”.

#### Returns

**index** [int] The index of the newly-added virtual site in the molecule

**add\_bond**(*self*, *atom1*, *atom2*, *bond\_order*, *is\_aromatic*, *stereochemistry*=None, *fractional\_bond\_order*=None)

Add a bond between two specified atom indices

#### Parameters

**atom1** [int or openforcefield.topology.molecule.Atom] Index of first atom

**atom2** [int or openforcefield.topology.molecule.Atom] Index of second atom

**bond\_order** [int] Integral bond order of Kekulized form

**is\_aromatic** [bool] True if this bond is aromatic, False otherwise

**stereochemistry** [str, optional, default=None] Either 'E' or 'Z' for specified stereochemistry, or None if stereochemistry is irrelevant

**fractional\_bond\_order** [float, optional, default=None] The fractional (eg. Wiberg) bond order

#### Returns

**index: int** Index of the bond in this molecule

**add\_conformer**(*self*, *coordinates*)

Add a conformation of the molecule

#### Parameters

**coordinates: simtk.unit.Quantity(np.array) with shape (n\_atoms, 3) and dimension of distance**

Coordinates of the new conformer, with the first dimension of the array corresponding to the atom index in the Molecule's indexing system.

#### Returns

**index: int** The index of this conformer

### openforcefield.topology.Topology

**class** openforcefield.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 simtk.openmm import app
>>> from openforcefield.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 openforcefield.topology 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)]
```

### Attributes

**angles** Iterable of Tuple[TopologyAtom]: iterator over the angles in this Topology.

**aromaticity\_model** Get the aromaticity model applied to all molecules in the topology.

**box\_vectors** Return the box vectors of the topology, if specified

**charge\_model** Get the partial charge model applied to all molecules in the topology.

**constrained\_atom\_pairs** Returns the constrained atom pairs of the Topology

**fractional\_bond\_order\_model** Get the fractional bond order model for the Topology.

**impropers** Iterable of Tuple[TopologyAtom]: iterator over the improper torsions in this Topology.

**n\_angles** int: number of angles in this Topology.

**n\_impropers** int: number of improper torsions in this Topology.

**n\_propers** int: number of proper torsions in this Topology.

**n\_reference\_molecules** Returns the number of reference (unique) molecules in in this Topology.

**n\_topology\_atoms** Returns the number of topology atoms in in this Topology.

**n\_topology\_bonds** Returns the number of TopologyBonds in in this Topology.

**n\_topology\_molecules** Returns the number of topology molecules in in this Topology.

**n\_topology\_particles** Returns the number of topology particles (TopologyAtoms and TopologyVirtualSites) in in this Topology.

**n\_topology\_virtual\_sites** Returns the number of TopologyVirtualSites in in this Topology.

**propers** Iterable of Tuple[TopologyAtom]: iterator over the proper torsions in this Topology.

**reference\_molecules** Get an iterator of reference molecules in this Topology.

**topology\_atoms** Returns an iterator over the atoms in this Topology.

**topology\_bonds** Returns an iterator over the bonds in this Topology

**topology\_molecules** Returns an iterator over all the TopologyMolecules in this Topology

**topology\_particles** Returns an iterator over the particles (TopologyAtoms and TopologyVirtualSites) in this Topology.

**topology\_virtual\_sites** Get an iterator over the virtual sites in this Topology

## Methods

<code>add_constraint(self, iatom, jatom[, distance])</code>	Mark a pair of atoms as constrained.
<code>add_molecule(self, molecule[, ...])</code>	Add a Molecule to the Topology.
<code>add_particle(self, particle)</code>	Add a Particle to the Topology.
<code>assert_bonded(self, atom1, atom2)</code>	Raise an exception if the specified atoms are not bonded in the topology.
<code>atom(self, atom_topology_index)</code>	Get the TopologyAtom at a given Topology atom index.
<code>bond(self, bond_topology_index)</code>	Get the TopologyBond at a given Topology bond index.
<code>chemical_environment_matches(self, query[, ...])</code>	Retrieve all matches for a given chemical environment query.
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(d)</code>	Static constructor from dictionary representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_mdtraj(mdtraj_topology[, unique_molecules])</code>	Construct an openforcefield Topology object from an MDTraj Topology object.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_molecules(molecules)</code>	Create a new Topology object containing one copy of each of the specified molecule(s).
<code>from_openmm(openmm_topology[, unique_molecules])</code>	Construct an openforcefield Topology object from an OpenMM Topology object.
<code>from_parmed(parmed_structure[, unique_molecules])</code>	

**Warning:**

This functionality will be implemented in a future toolkit release.

<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
--------------------------------------	--

Continued on next page

Table 8 – continued from previous page

<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>get_bond_between(self, i, j)</code>	Returns the bond between two atoms
<code>get_fractional_bond_order(self, iatom, jatom)</code>	Retrieve the fractional bond order for a bond.
<code>is_bonded(self, i, j)</code>	Returns True if the two atoms are bonded
<code>is_constrained(self, iatom, jatom)</code>	Check if a pair of atoms are marked as constrained.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Convert to dictionary representation.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_openmm(self)</code>	Create an OpenMM Topology object.
<code>to_parmed(self)</code>	

**Warning:**

This functionality will be implemented in a future toolkit release.

<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.
<code>virtual_site(self, vsite_topology_index)</code>	Get the TopologyAtom at a given Topology atom index.

`__init__(self, other=None)`  
Create a new Topology.

**Parameters**

**other** [optional, default=None] If specified, attempt to construct a copy of the Topology from the specified object. This might be a Topology object, or a file that can be used to construct a Topology object or serialized Topology object.

## Methods

<code>__init__(self[, other])</code>	Create a new Topology.
<code>add_constraint(self, iatom, jatom[, distance])</code>	Mark a pair of atoms as constrained.
<code>add_molecule(self, molecule[, ...])</code>	Add a Molecule to the Topology.
<code>add_particle(self, particle)</code>	Add a Particle to the Topology.
<code>assert_bonded(self, atom1, atom2)</code>	Raise an exception if the specified atoms are not bonded in the topology.
<code>atom(self, atom_topology_index)</code>	Get the TopologyAtom at a given Topology atom index.
<code>bond(self, bond_topology_index)</code>	Get the TopologyBond at a given Topology bond index.
<code>chemical_environment_matches(self, query[, ...])</code>	Retrieve all matches for a given chemical environment query.
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(d)</code>	Static constructor from dictionary representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_mdtraj(mdtraj_topology[, unique_molecules])</code>	Construct an openforcefield Topology object from an MDTraj Topology object.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_molecules(molecules)</code>	Create a new Topology object containing one copy of each of the specified molecule(s).
<code>from_openmm(openmm_topology[, unique_molecules])</code>	Construct an openforcefield Topology object from an OpenMM Topology object.
<code>from_parmed(parmed_structure[, unique_molecules])</code>	
<div style="border: 1px solid black; padding: 5px; display: inline-block;"> <p><b>Warning:</b> This functionality will be implemented in a future toolkit release.</p> </div>	
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.

Continued on next page

Table 9 – continued from previous page

<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>get_bond_between(self, i, j)</code>	Returns the bond between two atoms
<code>get_fractional_bond_order(self, iatom, jatom)</code>	Retrieve the fractional bond order for a bond.
<code>is_bonded(self, i, j)</code>	Returns True if the two atoms are bonded
<code>is_constrained(self, iatom, jatom)</code>	Check if a pair of atoms are marked as constrained.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Convert to dictionary representation.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_openmm(self)</code>	Create an OpenMM Topology object.
<code>to_parmed(self)</code>	

**Warning:**

This functionality will be implemented in a future toolkit release.

<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.
<code>virtual_site(self, vsite_topology_index)</code>	Get the TopologyAtom at a given Topology atom index.

**Attributes**

<code>angles</code>	Iterable of Tuple[TopologyAtom]: iterator over the angles in this Topology.
<code>aromaticity_model</code>	Get the aromaticity model applied to all molecules in the topology.

Continued on next page



Table 10 – continued from previous page

<code>box_vectors</code>	Return the box vectors of the topology, if specified Returns <code>box_vectors</code> : <code>simtk.unit.Quantity</code> wrapped numpy array The unit-wrapped box vectors of this topology
<code>charge_model</code>	Get the partial charge model applied to all molecules in the topology.
<code>constrained_atom_pairs</code>	Returns the constrained atom pairs of the Topology
<code>fractional_bond_order_model</code>	Get the fractional bond order model for the Topology.
<code>impropers</code>	Iterable of <code>Tuple[TopologyAtom]</code> : iterator over the improper torsions in this Topology.
<code>n_angles</code>	int: number of angles in this Topology.
<code>n_impropers</code>	int: number of improper torsions in this Topology.
<code>n_propers</code>	int: number of proper torsions in this Topology.
<code>n_reference_molecules</code>	Returns the number of reference (unique) molecules in in this Topology.
<code>n_topology_atoms</code>	Returns the number of topology atoms in in this Topology.
<code>n_topology_bonds</code>	Returns the number of TopologyBonds in in this Topology.
<code>n_topology_molecules</code>	Returns the number of topology molecules in in this Topology.
<code>n_topology_particles</code>	Returns the number of topology particles (TopologyAtoms and TopologyVirtualSites) in in this Topology.
<code>n_topology_virtual_sites</code>	Returns the number of TopologyVirtualSites in in this Topology.
<code>propers</code>	Iterable of <code>Tuple[TopologyAtom]</code> : iterator over the proper torsions in this Topology.
<code>reference_molecules</code>	Get an iterator of reference molecules in this Topology.
<code>topology_atoms</code>	Returns an iterator over the atoms in this Topology.
<code>topology_bonds</code>	Returns an iterator over the bonds in this Topology
<code>topology_molecules</code>	Returns an iterator over all the Topology-Molecules in this Topology
<code>topology_particles</code>	Returns an iterator over the particles (TopologyAtoms and TopologyVirtualSites) in this Topology.
<code>topology_virtual_sites</code>	Get an iterator over the virtual sites in this Topology

**property `reference_molecules`**

Get an iterator of reference molecules in this Topology.

**Returns****iterable of `openforcefield.topology.Molecule`****classmethod `from_molecules(molecules)`**

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

**Parameters**

**molecules** [Molecule or iterable of Molecules] One or more molecules to be added to the Topology

**Returns**

**topology** [Topology] The Topology created from the specified molecule(s)

**assert\_bonded**(*self*, *atom1*, *atom2*)

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

**Parameters**

**atom1**, **atom2** [openforcefield.topology.Atom or int] The atoms or atom topology indices to check to ensure they are bonded

**property aromaticity\_model**

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

**Returns**

**aromaticity\_model** [str] Aromaticity model in use.

**property box\_vectors**

Return the box vectors of the topology, if specified Returns — box\_vectors : simtk.unit.Quantity wrapped numpy array

The unit-wrapped box vectors of this topology

**property charge\_model**

Get the partial charge model applied to all molecules in the topology.

**Returns**

**charge\_model** [str] Charge model used for all molecules in the Topology.

**property constrained\_atom\_pairs**

Returns the constrained atom pairs of the Topology

**Returns**

**constrained\_atom\_pairs** [dict] dictionary of the form d[(atom1\_topology\_index, atom2\_topology\_index)] = distance (float)

**property fractional\_bond\_order\_model**

Get the fractional bond order model for the Topology.

**Returns**

**fractional\_bond\_order\_model** [str] Fractional bond order model in use.

**property n\_reference\_molecules**

Returns the number of reference (unique) molecules in in this Topology.

**Returns**

**n\_reference\_molecules** [int]

**property n\_topology\_molecules**

Returns the number of topology molecules in in this Topology.

**Returns**

**n\_topology\_molecules** [int]

**property topology\_molecules**

Returns an iterator over all the TopologyMolecules in this Topology

**Returns**

**topology\_molecules** [Iterable of TopologyMolecule]

**property n\_topology\_atoms**

Returns the number of topology atoms in in this Topology.

**Returns**

**n\_topology\_atoms** [int]

**property topology\_atoms**

Returns an iterator over the atoms in this Topology. These will be in ascending order of topology index (Note that this is not necessarily the same as the reference molecule index)

**Returns**

**topology\_atoms** [Iterable of TopologyAtom]

**property n\_topology\_bonds**

Returns the number of TopologyBonds in in this Topology.

**Returns**

**n\_bonds** [int]

**property topology\_bonds**

Returns an iterator over the bonds in this Topology

**Returns**

**topology\_bonds** [Iterable of TopologyBond]

**property n\_topology\_particles**

Returns the number of topology particles (TopologyAtoms and TopologyVirtualSites) in in this Topology.

**Returns**

**n\_topology\_particles** [int]

**property topology\_particles**

Returns an iterator over the particles (TopologyAtoms and TopologyVirtualSites) in this Topology. The TopologyAtoms will be in order of ascending Topology index (Note that this may differ from the order of atoms in the reference molecule index).

**Returns**

**topology\_particles** [Iterable of TopologyAtom and TopologyVirtualSite]

**property n\_topology\_virtual\_sites**

Returns the number of TopologyVirtualSites in in this Topology.

**Returns**

**n\_virtual\_sites** [iterable of TopologyVirtualSites]

**property topology\_virtual\_sites**

Get an iterator over the virtual sites in this Topology

**Returns**

**topology\_virtual\_sites** [Iterable of TopologyVirtualSite]

**property n\_angles**

int: number of angles in this Topology.

**property angles**

Iterable of Tuple[TopologyAtom]: iterator over the angles in this Topology.

**property n\_propers**

int: number of proper torsions in this Topology.

**property propers**

Iterable of Tuple[TopologyAtom]: iterator over the proper torsions in this Topology.

**property n\_impropers**

int: number of improper torsions in this Topology.

**property impropers**

Iterable of Tuple[TopologyAtom]: iterator over the improper torsions in this Topology.

**chemical\_environment\_matches**(*self*, *query*, *aromaticity\_model*='MDL',  
*toolkit\_registry*=<openforcefield.utils.toolkits.ToolkitRegistry  
object at 0x7f769344efd0>)

Retrieve all matches for a given chemical environment query.

TODO: \* Do we want to generalize this to other kinds of queries too, like mdtraj DSL, pymol selections, atom index slices, etc?

We could just call it topology.matches(query)

**Parameters**

**query** [str or ChemicalEnvironment] SMARTS string (with one or more tagged atoms) or ChemicalEnvironment query. Query will internally be resolved to SMARTS using query.as\_smarts() if it has an .as\_smarts method.

**aromaticity\_model** [str] Override the default aromaticity model for this topology and use the specified aromaticity model instead. Allowed values: ['MDL']

**Returns**

**matches** [list of Topology\_ChemicalEnvironmentMatch] A list of tuples, containing the topology indices of the matching atoms.

**to\_dict(self)**

Convert to dictionary representation.

**classmethod from\_dict(d)**

Static constructor from dictionary representation.

**classmethod from\_openmm(openmm\_topology, unique\_molecules=None)**

Construct an openforcefield Topology object from an OpenMM Topology object.

**Parameters**

**openmm\_topology** [simtk.openmm.app.Topology] An OpenMM Topology object

**unique\_molecules** [iterable of objects that can be used to construct unique Molecule objects] All unique molecules must be provided, in any order, though multiple copies of each molecule are allowed. The atomic elements and bond connectivity will be used to match the reference molecules to molecule graphs appearing in the OpenMM Topology. If bond orders are present in the OpenMM Topology, these will be used in matching as well. If all bonds have bond orders assigned in

`mdtraj_topology`, these bond orders will be used to attempt to construct the list of unique Molecules if the `unique_molecules` argument is omitted.

#### Returns

**topology** [`openforcefield.topology.Topology`] An openforcefield Topology object

**to\_openmm**(*self*)

Create an OpenMM Topology object.

The OpenMM Topology object will have one residue per topology molecule. Currently, the number of chains depends on how many copies of the same molecule are in the Topology. Molecules with more than 5 copies are all assigned to a single chain, otherwise one chain is created for each molecule. This behavior may change in the future.

#### Parameters

**openmm\_topology** [`simtk.openmm.app.Topology`] An OpenMM Topology object

**static from\_mdtraj**(*mdtraj\_topology*, *unique\_molecules=None*)

Construct an openforcefield Topology object from an MDTraj Topology object.

#### Parameters

**mdtraj\_topology** [`mdtraj.Topology`] An MDTraj Topology object

**unique\_molecules** [iterable of objects that can be used to construct unique Molecule objects] All unique molecules must be provided, in any order, though multiple copies of each molecule are allowed. The atomic elements and bond connectivity will be used to match the reference molecules to molecule graphs appearing in the MDTraj Topology. If bond orders are present in the MDTraj Topology, these will be used in matching as well. If all bonds have bond orders assigned in `mdtraj_topology`, these bond orders will be used to attempt to construct the list of unique Molecules if the `unique_molecules` argument is omitted.

#### Returns

**topology** [`openforcefield.Topology`] An openforcefield Topology object

**static from\_parmed**(*parmed\_structure*, *unique\_molecules=None*)

**Warning:** This functionality will be implemented in a future toolkit release.

Construct an openforcefield Topology object from a ParmEd Structure object.

#### Parameters

**parmed\_structure** [`parmed.Structure`] A ParmEd structure object

**unique\_molecules** [iterable of objects that can be used to construct unique Molecule objects] All unique molecules must be provided, in any order, though multiple copies of each molecule are allowed. The atomic elements and bond connectivity will be used to match the reference molecules to molecule graphs appearing in the structure's topology object. If bond orders are present in the structure's topology object, these will be used in matching as well. If all bonds have bond orders assigned in the structure's topology object, these bond orders will be used to attempt to construct the list of unique Molecules if the `unique_molecules` argument is omitted.

#### Returns

**topology** [openforcefield.Topology] An openforcefield Topology object

**to\_parmed**(*self*)

**Warning:** This functionality will be implemented in a future toolkit release.

Create a ParmEd Structure object.

**Returns**

**parmed\_structure** [parmed.Structure] A ParmEd Structure object

**get\_bond\_between**(*self*, *i*, *j*)

Returns the bond between two atoms

**Parameters**

**i, j** [int or TopologyAtom] Atoms or atom indices to check

**Returns**

**bond** [TopologyBond] The bond between *i* and *j*.

**is\_bonded**(*self*, *i*, *j*)

Returns True if the two atoms are bonded

**Parameters**

**i, j** [int or TopologyAtom] Atoms or atom indices to check

**Returns**

**is\_bonded** [bool] True if atoms are bonded, False otherwise.

**atom**(*self*, *atom\_topology\_index*)

Get the TopologyAtom at a given Topology atom index.

**Parameters**

**atom\_topology\_index** [int] The index of the TopologyAtom in this Topology

**Returns**

An openforcefield.topology.TopologyAtom

**virtual\_site**(*self*, *vsite\_topology\_index*)

Get the TopologyAtom at a given Topology atom index.

**Parameters**

**vsite\_topology\_index** [int] The index of the TopologyVirtualSite in this Topology

**Returns**

An openforcefield.topology.TopologyVirtualSite

**bond**(*self*, *bond\_topology\_index*)

Get the TopologyBond at a given Topology bond index.

**Parameters**

**bond\_topology\_index** [int] The index of the TopologyBond in this Topology

**Returns**

**An openforcefield.topology.TopologyBond****add\_particle**(*self*, *particle*)

Add a Particle to the Topology.

**Parameters****particle** [Particle] The Particle to be added. The Topology will take ownership of the Particle.**add\_molecule**(*self*, *molecule*, *local\_topology\_to\_reference\_index=None*)

Add a Molecule to the Topology.

**Parameters****molecule** [Molecule] The Molecule to be added.**local\_topology\_to\_reference\_index**: dict, optional, default = None Dictionary of {TopologyMolecule\_atom\_index : Molecule\_atom\_index} for the Topology-Molecule that will be built**Returns****index** [int] The index of this molecule in the topology**classmethod from\_bson**(*serialized*)

Instantiate an object from a BSON serialized representation.

Specification: <http://bsonspec.org/>**Parameters****serialized** [bytes] A BSON serialized representation of the object**Returns****instance** [cls] An instantiated object**classmethod from\_json**(*serialized*)

Instantiate an object from a JSON serialized representation.

Specification: <https://www.json.org/>**Parameters****serialized** [str] A JSON serialized representation of the object**Returns****instance** [cls] An instantiated object**classmethod from\_messagepack**(*serialized*)

Instantiate an object from a MessagePack serialized representation.

Specification: <https://msgpack.org/index.html>**Parameters****serialized** [bytes] A MessagePack-encoded bytes serialized representation**Returns****instance** [cls] Instantiated object.**classmethod from\_pickle**(*serialized*)

Instantiate an object from a pickle serialized representation.

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

**Parameters**

**serialized** [str] A pickled representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_toml**(*serialized*)

Instantiate an object from a TOML serialized representation.

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

**Parameters**

**serialized** [str] A TOML serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_xml**(*serialized*)

Instantiate an object from an XML serialized representation.

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

**Parameters**

**serialized** [bytes] An XML serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod from\_yaml**(*serialized*)

Instantiate from a YAML serialized representation.

Specification: <http://yaml.org/>

**Parameters**

**serialized** [str] A YAML serialized representation of the object

**Returns**

**instance** [cls] Instantiated object

**to\_bson**(*self*)

Return a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Returns**

**serialized** [bytes] A BSON serialized representation of the object

**to\_json**(*self*, *indent=None*)

Return a JSON serialized representation.

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

**Parameters**



**indent** [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

#### Returns

**serialized** [str] A JSON serialized representation of the object

**to\_messagepack(self)**

Return a MessagePack representation.

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

#### Returns

**serialized** [bytes] A MessagePack-encoded bytes serialized representation of the object

**to\_pickle(self)**

Return a pickle serialized representation.

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

#### Returns

**serialized** [str] A pickled representation of the object

**to\_toml(self)**

Return a TOML serialized representation.

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

#### Returns

**serialized** [str] A TOML serialized representation of the object

**to\_xml(self, indent=2)**

Return an XML representation.

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

#### Parameters

**indent** [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

#### Returns

**serialized** [bytes] A MessagePack-encoded bytes serialized representation.

**to\_yaml(self)**

Return a YAML serialized representation.

Specification: <http://yaml.org/>

#### Returns

**serialized** [str] A YAML serialized representation of the object

**add\_constraint(self, 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, jatom** [Atom] Atoms to mark as constrained These atoms may be bonded or not in the Topology

**distance** [simtk.unit.Quantity, optional, default=True] Constraint distance True if distance has yet to be determined False if constraint is to be removed

**is\_constrained**(*self*, *iatom*, *jatom*)

Check if a pair of atoms are marked as constrained.

**Parameters**

**iatom, jatom** [int] Indices of atoms to mark as constrained.

**Returns**

**distance** [simtk.unit.Quantity or bool] True if constrained but constraints have not yet been applied Distance if constraint has already been added to System

**get\_fractional\_bond\_order**(*self*, *iatom*, *jatom*)

Retrieve the fractional bond order for a bond.

An Exception is raised if it cannot be determined.

**Parameters**

**iatom, jatom** [Atom] Atoms for which a fractional bond order is to be retrieved.

**Returns**

**order** [float] Fractional bond order between the two specified atoms.

**openforcefield.topology.TopologyMolecule**

**class** openforcefield.topology.**TopologyMolecule**(*reference\_molecule*, *topology*, *local\_topology\_to\_reference\_index=None*)

TopologyMolecules are built to be an efficient way to store large numbers of copies of the same molecule for parameterization and system preparation.

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

**Attributes**

**angles** Iterable of Tuple[TopologyAtom]: iterator over the angles in this Topology.

**atom\_start\_topology\_index** Get the topology index of the first atom in this Topology-Molecule

**atoms** Return an iterator of all the TopologyAtoms in this TopologyMolecule

**bond\_start\_topology\_index** Get the topology index of the first bond in this Topology-Molecule

**bonds** Return an iterator of all the TopologyBonds in this TopologyMolecule

**impropers** Iterable of Tuple[TopologyAtom]: iterator over the improper torsions in this Topology.

**n\_angles** int: number of angles in this Topology.

**n\_atoms** The number of atoms in this topology.

**n\_bonds** Get the number of bonds in this TopologyMolecule

**n\_impropers** int: number of proper torsions in this Topology.

**n\_particles** Get the number of particles in this TopologyMolecule

**n\_propers** int: number of proper torsions in this Topology.

**n\_virtual\_sites** Get the number of virtual sites in this TopologyMolecule

**particle\_start\_topology\_index** Get the topology index of the first particle in this TopologyMolecule.

**particles** Return an iterator of all the TopologyParticles in this TopologyMolecules

**propers** Iterable of Tuple[TopologyAtom]: iterator over the proper torsions in this Topology.

**reference\_molecule** Get the reference molecule for this TopologyMolecule

**topology** Get the topology that this TopologyMolecule belongs to

**virtual\_site\_start\_topology\_index** Get the topology index of the first virtual site in this TopologyMolecule

**virtual\_sites** Return an iterator of all the TopologyVirtualSites in this Topology-Molecules

## Methods

<code>atom(self, index)</code>	Get the TopologyAtom with a given topology atom index in this TopologyMolecule.
<code>bond(self, index)</code>	Get the TopologyBond with a given reference molecule index in this TopologyMolecule
<code>from_dict(d)</code>	Static constructor from dictionary representation.
<code>particle(self, index)</code>	Get the TopologyParticle with a given reference molecule index in this TopologyMolecule
<code>to_dict(self)</code>	Convert to dictionary representation.
<code>virtual_site(self, index)</code>	Get the TopologyVirtualSite with a given reference molecule index in this TopologyMolecule

**`__init__(self, reference_molecule, topology, local_topology_to_reference_index=None)`**  
Create a new TopologyMolecule.

## Parameters

**reference\_molecule** [an openforcefield.topology.molecule.Molecule] The reference molecule, with details like formal charges, partial charges, bond orders, partial bond orders, and atomic symbols.

**topology** [an openforcefield.topology.Topology] The topology that this Topology-Molecule belongs to

**local\_topology\_to\_reference\_index** [dict, optional, default=None] Dictionary of {TopologyMolecule\_atom\_index : Molecule\_atom\_index} for the Topology-Molecule that will be built

## Methods

<code>__init__(self, reference_molecule, topology)</code>	Create a new TopologyMolecule.
<code>atom(self, index)</code>	Get the TopologyAtom with a given topology atom index in this TopologyMolecule.
<code>bond(self, index)</code>	Get the TopologyBond with a given reference molecule index in this TopologyMolecule
<code>from_dict(d)</code>	Static constructor from dictionary representation.
<code>particle(self, index)</code>	Get the TopologyParticle with a given reference molecule index in this TopologyMolecule
<code>to_dict(self)</code>	Convert to dictionary representation.
<code>virtual_site(self, index)</code>	Get the TopologyVirtualSite with a given reference molecule index in this TopologyMolecule

## Attributes

<code>angles</code>	Iterable of Tuple[TopologyAtom]: iterator over the angles in this Topology.
<code>atom_start_topology_index</code>	Get the topology index of the first atom in this TopologyMolecule
<code>atoms</code>	Return an iterator of all the TopologyAtoms in this TopologyMolecule
<code>bond_start_topology_index</code>	Get the topology index of the first bond in this TopologyMolecule
<code>bonds</code>	Return an iterator of all the TopologyBonds in this TopologyMolecule
<code>impropers</code>	Iterable of Tuple[TopologyAtom]: iterator over the improper torsions in this Topology.
<code>n_angles</code>	int: number of angles in this Topology.
<code>n_atoms</code>	The number of atoms in this topology.
<code>n_bonds</code>	Get the number of bonds in this Topology-Molecule
<code>n_impropers</code>	int: number of proper torsions in this Topology.
<code>n_particles</code>	Get the number of particles in this Topology-Molecule
<code>n_propers</code>	int: number of proper torsions in this Topology.
<code>n_virtual_sites</code>	Get the number of virtual sites in this Topology-Molecule
<code>particle_start_topology_index</code>	Get the topology index of the first particle in this TopologyMolecule.
<code>particles</code>	Return an iterator of all the TopologyParticles in this TopologyMolecules
<code>propers</code>	Iterable of Tuple[TopologyAtom]: iterator over the proper torsions in this Topology.
<code>reference_molecule</code>	Get the reference molecule for this Topology-Molecule
<code>topology</code>	Get the topology that this TopologyMolecule belongs to

Continued on next page

Table 13 – continued from previous page

<code>virtual_site_start_topology_index</code>	Get the topology index of the first virtual site in this TopologyMolecule
<code>virtual_sites</code>	Return an iterator of all the TopologyVirtualSites in this TopologyMolecules

**property topology**

Get the topology that this TopologyMolecule belongs to

**Returns**

an `openforcefield.topology.Topology`

**property reference\_molecule**

Get the reference molecule for this TopologyMolecule

**Returns**

an `openforcefield.topology.molecule.Molecule`

**property n\_atoms**

The number of atoms in this topology.

**Returns**

`int`

**atom(*self*, *index*)**

Get the TopologyAtom with a given topology atom index in this TopologyMolecule.

**Parameters**

**index** [`int`] Index of the TopologyAtom within this TopologyMolecule to retrieve

**Returns**

an `openforcefield.topology.TopologyAtom`

**property atoms**

Return an iterator of all the TopologyAtoms in this TopologyMolecule

**Returns**

an iterator of `openforcefield.topology.TopologyAtoms`

**property atom\_start\_topology\_index**

Get the topology index of the first atom in this TopologyMolecule

**bond(*self*, *index*)**

Get the TopologyBond with a given reference molecule index in this TopologyMolecule

**Parameters**

**index** [`int`] Index of the TopologyBond within this TopologyMolecule to retrieve

**Returns**

an `openforcefield.topology.TopologyBond`

**property bonds**

Return an iterator of all the TopologyBonds in this TopologyMolecule

**Returns**

an iterator of `openforcefield.topology.TopologyBonds`

**property n\_bonds**

Get the number of bonds in this TopologyMolecule

**Returns**

**int** [number of bonds]

**property bond\_start\_topology\_index**

Get the topology index of the first bond in this TopologyMolecule

**particle(self, index)**

Get the TopologyParticle with a given reference molecule index in this TopologyMolecule

**Parameters**

**index** [int] Index of the TopologyParticle within this TopologyMolecule to retrieve

**Returns**

**an openforcefield.topology.TopologyParticle**

**property particles**

Return an iterator of all the TopologyParticles in this TopologyMolecules

**Returns**

**an iterator of openforcefield.topology.TopologyParticle**

**property n\_particles**

Get the number of particles in this TopologyMolecule

**Returns**

**int** [The number of particles]

**property particle\_start\_topology\_index**

Get the topology index of the first particle in this TopologyMolecule.

**virtual\_site(self, index)**

Get the TopologyVirtualSite with a given reference molecule index in this TopologyMolecule

**Parameters**

**index** [int] Index of the TopologyVirtualSite within this TopologyMolecule to retrieve

**Returns**

**an openforcefield.topology.TopologyVirtualSite**

**property virtual\_sites**

Return an iterator of all the TopologyVirtualSites in this TopologyMolecules

**Returns**

**an iterator of openforcefield.topology.TopologyVirtualSite**

**property n\_virtual\_sites**

Get the number of virtual sites in this TopologyMolecule

**Returns**

**int**

**property angles**

Iterable of Tuple[TopologyAtom]: iterator over the angles in this Topology.

**property n\_angles**

int: number of angles in this Topology.

**property** `propers`

Iterable of `Tuple[TopologyAtom]`: iterator over the proper torsions in this `Topology`.

**property** `n_propers`

int: number of proper torsions in this `Topology`.

**property** `impropers`

Iterable of `Tuple[TopologyAtom]`: iterator over the improper torsions in this `Topology`.

**property** `n_impropers`

int: number of proper torsions in this `Topology`.

**property** `virtual_site_start_topology_index`

Get the topology index of the first virtual site in this `TopologyMolecule`

**to\_dict**(*self*)

Convert to dictionary representation.

**classmethod** `from_dict`(*d*)

Static constructor from dictionary representation.

## 2.1.2 Secondary objects

<code>Particle</code>	Base class for all particles in a molecule.
<code>Atom</code>	A particle representing a chemical atom.
<code>Bond</code>	Chemical bond representation.
<code>VirtualSite</code>	A particle representing a virtual site whose position is defined in terms of <code>Atom</code> positions.
<code>TopologyAtom</code>	A <code>TopologyAtom</code> is a lightweight data structure that represents a single openforcefield.topology.molecule.Atom in a <code>Topology</code> .
<code>TopologyBond</code>	A <code>TopologyBond</code> is a lightweight data structure that represents a single openforcefield.topology.molecule.Bond in a <code>Topology</code> .
<code>TopologyVirtualSite</code>	A <code>TopologyVirtualSite</code> is a lightweight data structure that represents a single openforcefield.topology.molecule.VirtualSite in a <code>Topology</code> .

### openforcefield.topology.Particle

**class** openforcefield.topology.`Particle`

Base class for all particles in a molecule.

A particle object could be an `Atom` or a `VirtualSite`.

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

**Attributes**

`molecule` The `Molecule` this atom is part of.

`molecule_particle_index` Returns the index of this particle in its molecule

`name` The name of the particle

## Methods

<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(d)</code>	Static constructor from dictionary representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Convert to dictionary representation.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

`__init__(self, /, *args, **kwargs)`  
Initialize self. See `help(type(self))` for accurate signature.

## Methods

<code>__init__(self, /, \*args, \*\*kwargs)</code>	Initialize self.
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(d)</code>	Static constructor from dictionary representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.

Continued on next page



Table 16 – continued from previous page

<code>to_dict(self)</code>	Convert to dictionary representation.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

**Attributes**

<code>molecule</code>	The Molecule this atom is part of.
<code>molecule_particle_index</code>	Returns the index of this particle in its molecule
<code>name</code>	The name of the particle

**property molecule**

The Molecule this atom is part of.

**property molecule\_particle\_index**

Returns the index of this particle in its molecule

**property name**

The name of the particle

**to\_dict(self)**

Convert to dictionary representation.

**classmethod from\_dict(d)**

Static constructor from dictionary representation.

**classmethod from\_bson(serialized)**

Instantiate an object from a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Parameters**

**serialized** [bytes] A BSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_json(serialized)**

Instantiate an object from a JSON serialized representation.

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

**Parameters**

**serialized** [str] A JSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_messagepack(serialized)**

Instantiate an object from a MessagePack serialized representation.

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

**Parameters**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod from\_pickle**(*serialized*)

Instantiate an object from a pickle serialized representation.

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

**Parameters**

**serialized** [str] A pickled representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_toml**(*serialized*)

Instantiate an object from a TOML serialized representation.

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

**Parameters**

**serialized** [str] A TOML serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_xml**(*serialized*)

Instantiate an object from an XML serialized representation.

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

**Parameters**

**serialized** [bytes] An XML serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod from\_yaml**(*serialized*)

Instantiate from a YAML serialized representation.

Specification: <http://yaml.org/>

**Parameters**

**serialized** [str] A YAML serialized representation of the object

**Returns**

**instance** [cls] Instantiated object

**to\_bson**(*self*)

Return a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Returns**

**serialized** [bytes] A BSON serialized representation of the object

**to\_json**(*self*, *indent=None*)

Return a JSON serialized representation.

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

#### Parameters

**indent** [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

#### Returns

**serialized** [str] A JSON serialized representation of the object

**to\_messagepack**(*self*)

Return a MessagePack representation.

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

#### Returns

**serialized** [bytes] A MessagePack-encoded bytes serialized representation of the object

**to\_pickle**(*self*)

Return a pickle serialized representation.

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

#### Returns

**serialized** [str] A pickled representation of the object

**to\_toml**(*self*)

Return a TOML serialized representation.

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

#### Returns

**serialized** [str] A TOML serialized representation of the object

**to\_xml**(*self*, *indent=2*)

Return an XML representation.

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

#### Parameters

**indent** [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

#### Returns

**serialized** [bytes] A MessagePack-encoded bytes serialized representation.

**to\_yaml**(*self*)

Return a YAML serialized representation.

Specification: <http://yaml.org/>

### Returns

**serialized** [str] A YAML serialized representation of the object

### openforcefield.topology.Atom

**class** openforcefield.topology.**Atom**(*atomic\_number*, *formal\_charge*, *is\_aromatic*, *name=None*,  
*molecule=None*, *stereochemistry=None*)

A particle representing a chemical atom.

Note that non-chemical virtual sites are represented by the `VirtualSite` object.

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

### 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.

**element** The element name

**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.

**molecule** The `Molecule` this atom is part of.

**molecule\_atom\_index** The index of this `Atom` within the the list of atoms in `Molecules`.

**molecule\_particle\_index** The index of this `Atom` within the the list of particles in the parent `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`)

**virtual\_sites** The list of `VirtualSite` objects this atom is involved in.

### Methods

<code>add_bond(self, bond)</code>	Adds a bond that this atom is involved in ..
<code>add_virtual_site(self, vsite)</code>	Adds a bond that this atom is involved in ..
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(atom_dict)</code>	Create an <code>Atom</code> from a dict representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.

Continued on next page

Table 18 – continued from previous page

<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>is_bonded_to(self, atom2)</code>	Determine whether this atom is bound to another atom
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Return a dict representation of the atom.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

`__init__(self, atomic_number, formal_charge, is_aromatic, name=None, molecule=None, stereochemistry=None)`

Create an immutable Atom object.

Object is serializable and immutable.

#### Parameters

**atomic\_number** [int] Atomic number of the atom

**formal\_charge** [int] Formal charge of the atom

**is\_aromatic** [bool] If True, atom is aromatic; if False, not aromatic

**stereochemistry** [str, optional, default=None] Either 'R' or 'S' for specified stereochemistry, or None for ambiguous stereochemistry

**name** [str, optional, default=None] An optional name to be associated with the atom

#### 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

<code>__init__(self, atomic_number, formal_charge, ...)</code>	Create an immutable Atom object.
<code>add_bond(self, bond)</code>	Adds a bond that this atom is involved in ..

Continued on next page

Table 19 – continued from previous page

<code>add_virtual_site(self, vsite)</code>	Adds a bond that this atom is involved in ..
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(atom_dict)</code>	Create an Atom from a dict representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>is_bonded_to(self, atom2)</code>	Determine whether this atom is bound to another atom
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Return a dict representation of the atom.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

### Attributes

<code>atomic_number</code>	The integer atomic number of the atom.
<code>bonded_atoms</code>	The list of Atom objects this atom is involved in bonds with
<code>bonds</code>	The list of Bond objects this atom is involved in.
<code>element</code>	The element name
<code>formal_charge</code>	The atom's formal charge
<code>is_aromatic</code>	The atom's <code>is_aromatic</code> flag
<code>mass</code>	The standard atomic weight (abundance-weighted isotopic mass) of the atomic site.
<code>molecule</code>	The Molecule this atom is part of.
<code>molecule_atom_index</code>	The index of this Atom within the the list of atoms in Molecules.
<code>molecule_particle_index</code>	The index of this Atom within the the list of particles in the parent Molecule.
<code>name</code>	The name of this atom, if any
<code>partial_charge</code>	The partial charge of the atom, if any.
<code>stereochemistry</code>	The atom's stereochemistry (if defined, otherwise None)
<code>virtual_sites</code>	The list of VirtualSite objects this atom is involved in.

**add\_bond**(*self*, *bond*)

Adds a bond that this atom is involved in .. todo :: Is this how we want to keep records?

**Parameters**

**bond:** an `openforcefield.topology.molecule.Bond` A bond involving this atom

**add\_virtual\_site**(*self*, *vsite*)

Adds a bond that this atom is involved in .. todo :: Is this how we want to keep records?

**Parameters**

**bond:** an `openforcefield.topology.molecule.Bond` A bond involving this atom

**to\_dict**(*self*)

Return a dict representation of the atom.

**classmethod from\_dict**(*atom\_dict*)

Create an Atom from a dict representation.

**property formal\_charge**

The atom's formal charge

**property partial\_charge**

The partial charge of the atom, if any.

**Returns**

`simtk.unit.Quantity` 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 element**

The element name

**property atomic\_number**

The integer atomic number of the atom.

**property mass**

The standard atomic weight (abundance-weighted isotopic mass) of the atomic site.

TODO (from jeff): Are there atoms that have different chemical properties based on their isotopes?

**property name**

The name of this atom, if any

**property bonds**

The list of `Bond` objects this atom is involved in.

**property bonded\_atoms**

The list of `Atom` objects this atom is involved in bonds with

**is\_bonded\_to**(*self*, *atom2*)

Determine whether this atom is bound to another atom

**Parameters**

**atom2:** `openforcefield.topology.molecule.Atom` a different atom in the same molecule

**Returns**

**bool** Whether this atom is bound to atom2

**property virtual\_sites**

The list of VirtualSite objects this atom is involved in.

**property molecule\_atom\_index**

The index of this Atom within the the list of atoms in Molecules. Note that this can be different from molecule\_particle\_index.

**property molecule\_particle\_index**

The index of this Atom within the the list of particles in the parent Molecule. Note that this can be different from molecule\_atom\_index.

**classmethod from\_bson(serialized)**

Instantiate an object from a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Parameters**

**serialized** [bytes] A BSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_json(serialized)**

Instantiate an object from a JSON serialized representation.

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

**Parameters**

**serialized** [str] A JSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_messagepack(serialized)**

Instantiate an object from a MessagePack serialized representation.

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

**Parameters**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod from\_pickle(serialized)**

Instantiate an object from a pickle serialized representation.

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

**Parameters**

**serialized** [str] A pickled representation of the object

**Returns**



**instance** [cls] An instantiated object

**classmethod from\_toml**(*serialized*)

Instantiate an object from a TOML serialized representation.

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

**Parameters**

**serialized** [str] A TOML serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_xml**(*serialized*)

Instantiate an object from an XML serialized representation.

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

**Parameters**

**serialized** [bytes] An XML serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod from\_yaml**(*serialized*)

Instantiate from a YAML serialized representation.

Specification: <http://yaml.org/>

**Parameters**

**serialized** [str] A YAML serialized representation of the object

**Returns**

**instance** [cls] Instantiated object

**property molecule**

The Molecule this atom is part of.

**to\_bson**(*self*)

Return a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Returns**

**serialized** [bytes] A BSON serialized representation of the object

**to\_json**(*self*, *indent=None*)

Return a JSON serialized representation.

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

**Parameters**

**indent** [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

**Returns**

**serialized** [str] A JSON serialized representation of the object

**to\_messagepack**(*self*)

Return a MessagePack representation.

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

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation of the object

**to\_pickle**(*self*)

Return a pickle serialized representation.

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

**Returns**

**serialized** [str] A pickled representation of the object

**to\_toml**(*self*)

Return a TOML serialized representation.

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

**Returns**

**serialized** [str] A TOML serialized representation of the object

**to\_xml**(*self*, *indent*=2)

Return an XML representation.

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

**Parameters**

**indent** [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation.

**to\_yaml**(*self*)

Return a YAML serialized representation.

Specification: <http://yaml.org/>

**Returns**

**serialized** [str] A YAML serialized representation of the object

## openforcefield.topology.Bond

**class** openforcefield.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** [openforcefield.topology.Atom] Atoms involved in the bond

**bondtype** [int] Discrete bond type representation for the Open Forcefield aromaticity model TODO: Do we want to pin ourselves to a single standard aromaticity model?

**type** [str] String based bond type

**order** [int] Integral bond order

**fractional\_bond\_order** [float, optional] Fractional bond order, or None.

**.. warning :: This API is experimental and subject to change.**

### Methods

<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(molecule, d)</code>	Create a Bond from a dict representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Return a dict representation of the bond.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

**`__init__(self, atom1, atom2, bond_order, is_aromatic, fractional_bond_order=None, stereochemistry=None)`**  
Create a new chemical bond.

### Methods

<code>__init__(self, atom1, atom2, bond_order, ...)</code>	Create a new chemical bond.
--	-----------------------------

Continued on next page

Table 22 – continued from previous page

<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(molecule, d)</code>	Create a Bond from a dict representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Return a dict representation of the bond.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

### Attributes

<code>atom1</code>	
<code>atom1_index</code>	
<code>atom2</code>	
<code>atom2_index</code>	
<code>atoms</code>	
<code>bond_order</code>	
<code>fractional_bond_order</code>	
<code>is_aromatic</code>	
<code>molecule</code>	
<code>molecule_bond_index</code>	The index of this Bond within the the list of bonds in Molecules.
<code>stereochemistry</code>	

### `to_dict(self)`

Return a dict representation of the bond.

### **classmethod** `from_dict(molecule, d)`

Create a Bond from a dict representation.

### **property** `molecule_bond_index`

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

### **classmethod** `from_bson(serialized)`

Instantiate an object from a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Parameters**

**serialized** [bytes] A BSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_json(serialized)**

Instantiate an object from a JSON serialized representation.

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

**Parameters**

**serialized** [str] A JSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_messagepack(serialized)**

Instantiate an object from a MessagePack serialized representation.

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

**Parameters**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod from\_pickle(serialized)**

Instantiate an object from a pickle serialized representation.

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

**Parameters**

**serialized** [str] A pickled representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_toml(serialized)**

Instantiate an object from a TOML serialized representation.

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

**Parameters**

**serialized** [str] A TOML serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_xml(serialized)**

Instantiate an object from an XML serialized representation.

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

**Parameters**

**serialized** [bytes] An XML serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod** `from_yaml(serialized)`

Instantiate from a YAML serialized representation.

Specification: <http://yaml.org/>

**Parameters**

**serialized** [str] A YAML serialized representation of the object

**Returns**

**instance** [cls] Instantiated object

**to\_bson**(*self*)

Return a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Returns**

**serialized** [bytes] A BSON serialized representation of the object

**to\_json**(*self*, *indent=None*)

Return a JSON serialized representation.

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

**Parameters**

**indent** [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

**Returns**

**serialized** [str] A JSON serialized representation of the object

**to\_messagepack**(*self*)

Return a MessagePack representation.

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

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation of the object

**to\_pickle**(*self*)

Return a pickle serialized representation.

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

**Returns**

**serialized** [str] A pickled representation of the object

`to_toml(self)`

Return a TOML serialized representation.

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

#### Returns

**serialized** [str] A TOML serialized representation of the object

`to_xml(self, indent=2)`

Return an XML representation.

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

#### Parameters

**indent** [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

#### Returns

**serialized** [bytes] A MessagePack-encoded bytes serialized representation.

`to_yaml(self)`

Return a YAML serialized representation.

Specification: <http://yaml.org/>

#### Returns

**serialized** [str] A YAML serialized representation of the object

### openforcefield.topology.VirtualSite

**class** openforcefield.topology.**VirtualSite**(atoms, charge\_increments=None, epsilon=None, sigma=None, rmin\_half=None, name=None)

A particle representing a virtual site whose position is defined in terms of Atom positions.

Note that chemical atoms are represented by the Atom.

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

#### Attributes

**atoms** Atoms on whose position this VirtualSite depends.

**charge\_increments** Charges taken from this VirtualSite's atoms and given to the VirtualSite

**epsilon** The VdW epsilon term of this VirtualSite

**molecule** The Molecule this atom is part of.

**molecule\_particle\_index** The index of this VirtualSite within the the list of particles in the parent Molecule.

**molecule\_virtual\_site\_index** The index of this VirtualSite within the list of virtual sites within Molecule Note that this can be different from particle\_index.

**name** The name of this VirtualSite

**rmin\_half** The VdW rmin\_half term of this VirtualSite

**sigma** The VdW sigma term of this VirtualSite

**type** The type of this VirtualSite (returns the class name as string)

## Methods

<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(vsite_dict)</code>	Create a virtual site from a dict representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Return a dict representation of the virtual site.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

`__init__(self, atoms, charge_increments=None, epsilon=None, sigma=None, rmin_half=None, name=None)`  
Base class for VirtualSites

## Parameters

**atoms** [list of Atom of shape [N]] `atoms[index]` is the corresponding Atom for `weights[index]`

**charge\_increments** [list of floats of shape [N], optional, default=None] The amount of charge to remove from the VirtualSite's atoms and put in the VirtualSite. Indexing in this list should match the ordering in the atoms list. Default is None.

**sigma** [float, default=None] Sigma term for VdW properties of virtual site. Default is None.

**epsilon** [float] Epsilon term for VdW properties of virtual site. Default is None.

**rmin\_half** [float] `Rmin_half` term for VdW properties of virtual site. Default is None.

**name** [string or None, default=None] The name of this virtual site. Default is None.

**virtual\_site\_type** [str] Virtual site type.

**name** [str or None, default=None] The name of this virtual site. Default is None



## Methods

<code>__init__(self, atoms[, charge_increments, ...])</code>	Base class for VirtualSites
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(vsite_dict)</code>	Create a virtual site from a dict representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Return a dict representation of the virtual site.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

## Attributes

<code>atoms</code>	Atoms on whose position this VirtualSite depends.
<code>charge_increments</code>	Charges taken from this VirtualSite's atoms and given to the VirtualSite
<code>epsilon</code>	The VdW epsilon term of this VirtualSite
<code>molecule</code>	The Molecule this atom is part of.
<code>molecule_particle_index</code>	The index of this VirtualSite within the the list of particles in the parent Molecule.
<code>molecule_virtual_site_index</code>	The index of this VirtualSite within the list of virtual sites within Molecule Note that this can be different from <code>particle_index</code> .
<code>name</code>	The name of this VirtualSite
<code>rmin_half</code>	The VdW <code>rmin_half</code> term of this VirtualSite
<code>sigma</code>	The VdW sigma term of this VirtualSite
<code>type</code>	The type of this VirtualSite (returns the class name as string)

### `to_dict(self)`

Return a dict representation of the virtual site.

### `classmethod from_dict(vsite_dict)`

Create a virtual site from a dict representation.

**property molecule\_virtual\_site\_index**

The index of this VirtualSite within the list of virtual sites within Molecule. Note that this can be different from `particle_index`.

**property molecule\_particle\_index**

The index of this VirtualSite within the the list of particles in the parent Molecule. Note that this can be different from `molecule_virtual_site_index`.

**property atoms**

Atoms on whose position this VirtualSite depends.

**property charge\_increments**

Charges taken from this VirtualSite's atoms and given to the VirtualSite

**property epsilon**

The VdW epsilon term of this VirtualSite

**property sigma**

The VdW sigma term of this VirtualSite

**property rmin\_half**

The VdW `rmin_half` term of this VirtualSite

**property name**

The name of this VirtualSite

**property type**

The type of this VirtualSite (returns the class name as string)

**classmethod from\_bson(*serialized*)**

Instantiate an object from a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Parameters**

**serialized** [bytes] A BSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_json(*serialized*)**

Instantiate an object from a JSON serialized representation.

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

**Parameters**

**serialized** [str] A JSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_messagepack(*serialized*)**

Instantiate an object from a MessagePack serialized representation.

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

**Parameters**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod** `from_pickle(serialized)`

Instantiate an object from a pickle serialized representation.

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

**Parameters**

**serialized** [str] A pickled representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod** `from_toml(serialized)`

Instantiate an object from a TOML serialized representation.

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

**Parameters**

**serialized** [str] A TOML serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod** `from_xml(serialized)`

Instantiate an object from an XML serialized representation.

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

**Parameters**

**serialized** [bytes] An XML serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod** `from_yaml(serialized)`

Instantiate from a YAML serialized representation.

Specification: <http://yaml.org/>

**Parameters**

**serialized** [str] A YAML serialized representation of the object

**Returns**

**instance** [cls] Instantiated object

**property** `molecule`

The Molecule this atom is part of.

**to\_bson(self)**

Return a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Returns**

**serialized** [bytes] A BSON serialized representation of the object

**to\_json**(*self*, *indent=None*)

Return a JSON serialized representation.

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

**Parameters**

**indent** [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

**Returns**

**serialized** [str] A JSON serialized representation of the object

**to\_messagepack**(*self*)

Return a MessagePack representation.

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

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation of the object

**to\_pickle**(*self*)

Return a pickle serialized representation.

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

**Returns**

**serialized** [str] A pickled representation of the object

**to\_toml**(*self*)

Return a TOML serialized representation.

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

**Returns**

**serialized** [str] A TOML serialized representation of the object

**to\_xml**(*self*, *indent=2*)

Return an XML representation.

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

**Parameters**

**indent** [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation.

**to\_yaml**(*self*)

Return a YAML serialized representation.

Specification: <http://yaml.org/>

**Returns**

**serialized** [str] A YAML serialized representation of the object

## openforcefield.topology.TopologyAtom

**class** openforcefield.topology.**TopologyAtom**(*atom*, *topology\_molecule*)

A TopologyAtom is a lightweight data structure that represents a single openforcefield.topology.molecule.Atom in a Topology. A TopologyAtom consists of two references – One to its fully detailed “atom”, an openforcefield.topology.molecule.Atom, and another to its parent “topology\_molecule”, which occupies a spot in the parent Topology’s TopologyMolecule list.

As some systems can be very large, there is no always-existing representation of a TopologyAtom. They are created on demand as the user requests them.

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

### Attributes

- atom** Get the reference Atom for this TopologyAtom.
- atomic\_number** Get the atomic number of this atom
- molecule** Get the reference Molecule that this TopologyAtom belongs to.
- topology\_atom\_index** Get the index of this atom in its parent Topology.
- topology\_bonds** Get the TopologyBonds connected to this TopologyAtom.
- topology\_molecule** Get the TopologyMolecule that this TopologyAtom belongs to.
- topology\_particle\_index** Get the index of this particle in its parent Topology.

### Methods

<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(d)</code>	Static constructor from dictionary representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Convert to dictionary representation.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.

Continued on next page

Table 27 – continued from previous page

<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

`__init__(self, atom, topology_molecule)`  
Create a new TopologyAtom.

#### Parameters

**atom** [An openforcefield.topology.molecule.Atom] The reference atom

**topology\_molecule** [An openforcefield.topology.TopologyMolecule] The Topology-Molecule that this TopologyAtom belongs to

#### Methods

<code>__init__(self, atom, topology_molecule)</code>	Create a new TopologyAtom.
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(d)</code>	Static constructor from dictionary representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Convert to dictionary representation.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

#### Attributes

<code>atom</code>	Get the reference Atom for this TopologyAtom.
<code>atomic_number</code>	Get the atomic number of this atom
<code>molecule</code>	Get the reference Molecule that this Topology-Atom belongs to.
<code>topology_atom_index</code>	Get the index of this atom in its parent Topology.

Continued on next page

Table 29 – continued from previous page

<code>topology_bonds</code>	Get the TopologyBonds connected to this TopologyAtom.
<code>topology_molecule</code>	Get the TopologyMolecule that this TopologyAtom belongs to.
<code>topology_particle_index</code>	Get the index of this particle in its parent Topology.

**property atom**

Get the reference Atom for this TopologyAtom.

**Returns****an openforcefield.topology.molecule.Atom****property atomic\_number**

Get the atomic number of this atom

**Returns****int****property topology\_molecule**

Get the TopologyMolecule that this TopologyAtom belongs to.

**Returns****openforcefield.topology.TopologyMolecule****property molecule**

Get the reference Molecule that this TopologyAtom belongs to.

**Returns****openforcefield.topology.molecule.Molecule****property topology\_atom\_index**

Get the index of this atom in its parent Topology.

**Returns****int** The index of this atom in its parent topology.**property topology\_particle\_index**

Get the index of this particle in its parent Topology.

**Returns****int** The index of this atom in its parent topology.**property topology\_bonds**

Get the TopologyBonds connected to this TopologyAtom.

**Returns****iterator of openforcefield.topology.TopologyBonds****to\_dict(self)**

Convert to dictionary representation.

**classmethod from\_dict(d)**

Static constructor from dictionary representation.

**classmethod** `from_bson(serialized)`

Instantiate an object from a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Parameters**

**serialized** [bytes] A BSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod** `from_json(serialized)`

Instantiate an object from a JSON serialized representation.

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

**Parameters**

**serialized** [str] A JSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod** `from_messagepack(serialized)`

Instantiate an object from a MessagePack serialized representation.

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

**Parameters**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod** `from_pickle(serialized)`

Instantiate an object from a pickle serialized representation.

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

**Parameters**

**serialized** [str] A pickled representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod** `from_toml(serialized)`

Instantiate an object from a TOML serialized representation.

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

**Parameters**

**serialized** [str] A TOML serialized representation of the object

**Returns**

**instance** [cls] An instantiated object



**classmethod** `from_xml(serialized)`

Instantiate an object from an XML serialized representation.

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

**Parameters**

**serialized** [bytes] An XML serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod** `from_yaml(serialized)`

Instantiate from a YAML serialized representation.

Specification: <http://yaml.org/>

**Parameters**

**serialized** [str] A YAML serialized representation of the object

**Returns**

**instance** [cls] Instantiated object

**to\_bson(self)**

Return a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Returns**

**serialized** [bytes] A BSON serialized representation of the object

**to\_json(self, indent=None)**

Return a JSON serialized representation.

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

**Parameters**

**indent** [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

**Returns**

**serialized** [str] A JSON serialized representation of the object

**to\_messagepack(self)**

Return a MessagePack representation.

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

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation of the object

**to\_pickle(self)**

Return a pickle serialized representation.

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

**Returns**

**serialized** [str] A pickled representation of the object

**to\_toml**(*self*)

Return a TOML serialized representation.

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

**Returns**

**serialized** [str] A TOML serialized representation of the object

**to\_xml**(*self*, *indent=2*)

Return an XML representation.

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

**Parameters**

**indent** [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation.

**to\_yaml**(*self*)

Return a YAML serialized representation.

Specification: <http://yaml.org/>

**Returns**

**serialized** [str] A YAML serialized representation of the object

## openforcefield.topology.TopologyBond

**class** openforcefield.topology.**TopologyBond**(*bond*, *topology\_molecule*)

A TopologyBond is a lightweight data structure that represents a single openforcefield.topology.molecule.Bond in a Topology. A TopologyBond consists of two references – One to its fully detailed “bond”, an openforcefield.topology.molecule.Bond, and another to its parent “topology\_molecule”, which occupies a spot in the parent Topology’s TopologyMolecule list.

As some systems can be very large, there is no always-existing representation of a TopologyBond. They are created on demand as the user requests them.

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

**Attributes**

**atoms** Get the TopologyAtoms connected to this TopologyBond.

**bond** Get the reference Bond for this TopologyBond.

**bond\_order** Get the order of this TopologyBond.

**molecule** Get the reference Molecule that this TopologyBond belongs to.

**topology\_bond\_index** Get the index of this bond in its parent Topology.

**topology\_molecule** Get the TopologyMolecule that this TopologyBond belongs to.

## Methods

<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(d)</code>	Static constructor from dictionary representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Convert to dictionary representation.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

`__init__(self, bond, topology_molecule)`

### Parameters

**bond** [An `openforcefield.topology.molecule.Bond`] The reference bond.

**topology\_molecule** [An `openforcefield.topology.TopologyMolecule`] The TopologyMolecule that this TopologyBond belongs to.

## Methods

<code>__init__(self, bond, topology_molecule)</code>	
<b>Parameters</b>	
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(d)</code>	Static constructor from dictionary representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.

Continued on next page

Table 31 – continued from previous page

<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Convert to dictionary representation.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

**Attributes**

<code>atoms</code>	Get the TopologyAtoms connected to this TopologyBond.
<code>bond</code>	Get the reference Bond for this TopologyBond.
<code>bond_order</code>	Get the order of this TopologyBond.
<code>molecule</code>	Get the reference Molecule that this TopologyBond belongs to.
<code>topology_bond_index</code>	Get the index of this bond in its parent Topology.
<code>topology_molecule</code>	Get the TopologyMolecule that this TopologyBond belongs to.

**property bond**

Get the reference Bond for this TopologyBond.

**Returns**

**an openforcefield.topology.molecule.Bond**

**property topology\_molecule**

Get the TopologyMolecule that this TopologyBond belongs to.

**Returns**

**openforcefield.topology.TopologyMolecule**

**property topology\_bond\_index**

Get the index of this bond in its parent Topology.

**Returns**

**int** The index of this bond in its parent topology.

**property molecule**

Get the reference Molecule that this TopologyBond belongs to.

**Returns**

**openforcefield.topology.molecule.Molecule**

**property bond\_order**

Get the order of this TopologyBond.

**Returns**

**int** [bond order]

**property atoms**

Get the TopologyAtoms connected to this TopologyBond.

**Returns**

**iterator of openforcefield.topology.TopologyAtom**

**to\_dict(self)**

Convert to dictionary representation.

**classmethod from\_dict(d)**

Static constructor from dictionary representation.

**classmethod from\_bson(serialized)**

Instantiate an object from a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Parameters**

**serialized** [bytes] A BSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_json(serialized)**

Instantiate an object from a JSON serialized representation.

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

**Parameters**

**serialized** [str] A JSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_messagepack(serialized)**

Instantiate an object from a MessagePack serialized representation.

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

**Parameters**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod from\_pickle(serialized)**

Instantiate an object from a pickle serialized representation.

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

**Parameters**

**serialized** [str] A pickled representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_toml(serialized)**

Instantiate an object from a TOML serialized representation.

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

**Parameters**

**serialized** [str] A TOML serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_xml(serialized)**

Instantiate an object from an XML serialized representation.

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

**Parameters**

**serialized** [bytes] An XML serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod from\_yaml(serialized)**

Instantiate from a YAML serialized representation.

Specification: <http://yaml.org/>

**Parameters**

**serialized** [str] A YAML serialized representation of the object

**Returns**

**instance** [cls] Instantiated object

**to\_bson(self)**

Return a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Returns**

**serialized** [bytes] A BSON serialized representation of the object

**to\_json(self, indent=None)**

Return a JSON serialized representation.

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

**Parameters**

**indent** [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

**Returns**

**serialized** [str] A JSON serialized representation of the object

**to\_messagepack(self)**

Return a MessagePack representation.

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

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation of the object

**to\_pickle(self)**

Return a pickle serialized representation.

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

**Returns**

**serialized** [str] A pickled representation of the object

**to\_toml(self)**

Return a TOML serialized representation.

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

**Returns**

**serialized** [str] A TOML serialized representation of the object

**to\_xml(self, indent=2)**

Return an XML representation.

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

**Parameters**

**indent** [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation.

**to\_yaml(self)**

Return a YAML serialized representation.

Specification: <http://yaml.org/>

**Returns**

**serialized** [str] A YAML serialized representation of the object

## openforcefield.topology.TopologyVirtualSite

**class** openforcefield.topology.**TopologyVirtualSite**(*virtual\_site*, *topology\_molecule*)

A TopologyVirtualSite is a lightweight data structure that represents a single openforcefield.topology.molecule.VirtualSite in a Topology. A TopologyVirtualSite consists of two references – One to its fully detailed “VirtualSite”, an openforcefield.topology.molecule.VirtualSite, and another to its parent “topology\_molecule”, which occupies a spot in the parent Topology’s TopologyMolecule list.

As some systems can be very large, there is no always-existing representation of a `TopologyVirtualSite`. They are created on demand as the user requests them.

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

### Attributes

**atoms** Get the `TopologyAtoms` involved in this `TopologyVirtualSite`.

**molecule** Get the reference `Molecule` that this `TopologyVirtualSite` belongs to.

**topology\_molecule** Get the `TopologyMolecule` that this `TopologyVirtualSite` belongs to.

**topology\_particle\_index** Get the index of this particle in its parent `Topology`.

**topology\_virtual\_site\_index** Get the index of this virtual site in its parent `Topology`.

**type** Get the type of this virtual site

**virtual\_site** Get the reference `VirtualSite` for this `TopologyVirtualSite`.

### Methods

<code>atom(self, index)</code>	Get the atom at a specific index in this <code>TopologyVirtualSite</code>
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(d)</code>	Static constructor from dictionary representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Convert to dictionary representation.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

`__init__(self, virtual_site, topology_molecule)`

### Parameters

**virtual\_site** [An `openforcefield.topology.molecule.VirtualSite`] The reference virtual



site

**topology\_molecule** [An openforcefield.topology.TopologyMolecule] The Topology-Molecule that this TopologyVirtualSite belongs to

## Methods

<code>__init__(self, virtual_site, topology_molecule)</code>	
Parameters	
<code>atom(self, index)</code>	Get the atom at a specific index in this TopologyVirtualSite
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(d)</code>	Static constructor from dictionary representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	Convert to dictionary representation.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

## Attributes

<code>atoms</code>	Get the TopologyAtoms involved in this TopologyVirtualSite.
<code>molecule</code>	Get the reference Molecule that this TopologyVirtualSite belongs to.
<code>topology_molecule</code>	Get the TopologyMolecule that this TopologyVirtualSite belongs to.
<code>topology_particle_index</code>	Get the index of this particle in its parent Topology.
<code>topology_virtual_site_index</code>	Get the index of this virtual site in its parent Topology.
<code>type</code>	Get the type of this virtual site

Continued on next page

Table 35 – continued from previous page

<code>virtual_site</code>	Get the reference VirtualSite for this TopologyVirtualSite.
---------------------------	---

<b><code>atom(self, index)</code></b>	Get the atom at a specific index in this TopologyVirtualSite
<b>Parameters</b>	
<b>index</b> [int]	The index of the atom in the reference VirtualSite to retrieve
<b>Returns</b>	
<b>TopologyAtom</b>	

<b><code>property atoms</code></b>	Get the TopologyAtoms involved in this TopologyVirtualSite.
<b>Returns</b>	
<b>iterator of openforcefield.topology.TopologyAtom</b>	

<b><code>property virtual_site</code></b>	Get the reference VirtualSite for this TopologyVirtualSite.
<b>Returns</b>	
<b>an openforcefield.topology.molecule.VirtualSite</b>	

<b><code>property topology_molecule</code></b>	Get the TopologyMolecule that this TopologyVirtualSite belongs to.
<b>Returns</b>	
<b>openforcefield.topology.TopologyMolecule</b>	

<b><code>property topology_virtual_site_index</code></b>	Get the index of this virtual site in its parent Topology.
<b>Returns</b>	
<b>int</b>	The index of this virtual site in its parent topology.

<b><code>property topology_particle_index</code></b>	Get the index of this particle in its parent Topology.
<b>Returns</b>	
<b>int</b>	The index of this particle in its parent topology.

<b><code>property molecule</code></b>	Get the reference Molecule that this TopologyVirtualSite belongs to.
<b>Returns</b>	
<b>openforcefield.topology.molecule.Molecule</b>	

<b><code>property type</code></b>	Get the type of this virtual site
<b>Returns</b>	
<b>str</b>	[The class name of this virtual site]

<b><code>to_dict(self)</code></b>	Convert to dictionary representation.
-----------------------------------	---------------------------------------

**classmethod from\_dict(*d*)**

Static constructor from dictionary representation.

**classmethod from\_bson(*serialized*)**

Instantiate an object from a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Parameters**

**serialized** [bytes] A BSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_json(*serialized*)**

Instantiate an object from a JSON serialized representation.

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

**Parameters**

**serialized** [str] A JSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_messagepack(*serialized*)**

Instantiate an object from a MessagePack serialized representation.

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

**Parameters**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod from\_pickle(*serialized*)**

Instantiate an object from a pickle serialized representation.

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

**Parameters**

**serialized** [str] A pickled representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_toml(*serialized*)**

Instantiate an object from a TOML serialized representation.

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

**Parameters**

**serialized** [str] A TOML serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**classmethod from\_xml(serialized)**

Instantiate an object from an XML serialized representation.

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

**Parameters**

**serialized** [bytes] An XML serialized representation

**Returns**

**instance** [cls] Instantiated object.

**classmethod from\_yaml(serialized)**

Instantiate from a YAML serialized representation.

Specification: <http://yaml.org/>

**Parameters**

**serialized** [str] A YAML serialized representation of the object

**Returns**

**instance** [cls] Instantiated object

**to\_bson(self)**

Return a BSON serialized representation.

Specification: <http://bsonspec.org/>

**Returns**

**serialized** [bytes] A BSON serialized representation of the object

**to\_json(self, indent=None)**

Return a JSON serialized representation.

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

**Parameters**

**indent** [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

**Returns**

**serialized** [str] A JSON serialized representation of the object

**to\_messagepack(self)**

Return a MessagePack representation.

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

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation of the object

**to\_pickle(self)**

Return a pickle serialized representation.

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

#### Returns

**serialized** [str] A pickled representation of the object

**to\_toml**(*self*)

Return a TOML serialized representation.

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

#### Returns

**serialized** [str] A TOML serialized representation of the object

**to\_xml**(*self*, *indent*=2)

Return an XML representation.

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

#### Parameters

**indent** [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

#### Returns

**serialized** [bytes] A MessagePack-encoded bytes serialized representation.

**to\_yaml**(*self*)

Return a YAML serialized representation.

Specification: <http://yaml.org/>

#### Returns

**serialized** [str] A YAML serialized representation of the object

## 2.2 Forcefield typing tools

### 2.2.1 Chemical environments

Tools for representing and operating on chemical environments

**Warning:** This class is largely redundant with the same one in the Chemper package, and will likely be removed.

---

`ChemicalEnvironment`

Chemical environment abstract base class that matches an atom, bond, angle, etc.

---

## openforcefield.typing.chemistry.ChemicalEnvironment

**class** openforcefield.typing.chemistry.ChemicalEnvironment(*smirks=None, label=None, replacements=None, toolkit=None*)

Chemical environment abstract base class that matches an atom, bond, angle, etc.

**Warning:** This class is largely redundant with the same one in the Chemper package, and will likely be removed.

## Methods

<code>Atom([ORtypes, ANDtypes, index, ring])</code>	Atom representation, which may have some ORtypes and ANDtypes properties.
<code>Bond([ORtypes, ANDtypes])</code>	Bond representation, which may have ORtype and ANDtype descriptors.
<code>addAtom(self, bondToAtom[, bondORtypes, ...])</code>	Add an atom to the specified target atom.
<code>asSMIRKS(self[, smarts])</code>	Returns a SMIRKS representation of the chemical environment
<code>getAlphaAtoms(self)</code>	Returns a list of atoms alpha to any indexed atom
<code>getAlphaBonds(self)</code>	Returns a list of Bond objects that connect
<code>getAtoms(self)</code>	

## Returns

<code>getBetaAtoms(self)</code>	Returns a list of atoms beta to any indexed atom
<code>getBetaBonds(self)</code>	Returns a list of Bond objects that connect
<code>getBond(self, atom1, atom2)</code>	Get bond between two atoms
<code>getBondOrder(self, atom)</code>	Returns minimum bond order around a given atom 0 if atom has no neighbors aromatic bonds count as 1.5 any bond counts as 1.0
<code>getBonds(self[, atom])</code>	

## Parameters

<code>getComponentList(self, component_type[, ...])</code>	Returns a list of atoms or bonds matching the descriptor
<code>getIndexedAtoms(self)</code>	returns the list of Atom objects with an index
<code>getIndexedBonds(self)</code>	Returns a list of Bond objects that connect two indexed atoms
<code>getNeighbors(self, atom)</code>	Returns atoms that are bound to the given atom in the form of a list of Atom objects
<code>getType(self)</code>	Uses number of indexed atoms and bond connectivity to determine the type of chemical environment
<code>getUnindexedAtoms(self)</code>	returns a list of Atom objects that are not indexed
<code>getUnindexedBonds(self)</code>	Returns a list of Bond objects that connect
<code>getValence(self, atom)</code>	Returns the valence (number of neighboring atoms) around the given atom

Continued on next page

Table 37 – continued from previous page

<code>isAlpha(self, component)</code>	Takes an atom or bond and returns True if it is alpha to an indexed atom
<code>isBeta(self, component)</code>	Takes an atom or bond and returns True if it is beta to an indexed atom
<code>isIndexed(self, component)</code>	returns True if the atom or bond is indexed
<code>isUnindexed(self, component)</code>	returns True if the atom or bond is not indexed
<code>isValid(self[, smirks])</code>	Returns if the environment is valid, that is if it creates a parseable SMIRKS string.
<code>removeAtom(self, atom[, onlyEmpty])</code>	Remove the specified atom from the chemical environment.
<code>selectAtom(self[, descriptor])</code>	Select a random atom fitting the descriptor.
<code>selectBond(self[, descriptor])</code>	Select a random bond fitting the descriptor.
<code>validate(smirks[, ensure_valence_type, toolkit])</code>	Validate the provided SMIRKS string is valid, and if requested, tags atoms appropriate to the specified valence type.

`__init__(self, smirks=None, label=None, replacements=None, toolkit=None)`

Initialize a chemical environment abstract base class.

**smirks = string, optional** if smirks is not None, a chemical environment is built from the provided SMIRKS string

**label = anything, optional** intended to be used to label this chemical environment could be a string, int, or float, or anything

**replacements = list of lists, optional**, [substitution, smarts] form for parsing SMIRKS

**toolkit = string. Default = None** Either 'openeye' or 'rdkit', indicating the backend to use for chemical perception. If None, this function will use OpenEye if possible, otherwise will use RDKit.

## Methods

<code>__init__(self[, smirks, label, ...])</code>	Initialize a chemical environment abstract base class.
<code>addAtom(self, bondToAtom[, bondORtypes, ...])</code>	Add an atom to the specified target atom.
<code>asSMIRKS(self[, smarts])</code>	Returns a SMIRKS representation of the chemical environment
<code>getAlphaAtoms(self)</code>	Returns a list of atoms alpha to any indexed atom
<code>getAlphaBonds(self)</code>	Returns a list of Bond objects that connect
<code>getAtoms(self)</code>	

## Returns

<code>getBetaAtoms(self)</code>	Returns a list of atoms beta to any indexed atom
<code>getBetaBonds(self)</code>	Returns a list of Bond objects that connect
<code>getBond(self, atom1, atom2)</code>	Get bond between two atoms
<code>getBondOrder(self, atom)</code>	Returns minimum bond order around a given atom 0 if atom has no neighbors aromatic bonds count as 1.5 any bond counts as 1.0

Continued on next page

Table 38 – continued from previous page

<code>getBonds(self[, atom])</code>	
Parameters	
<code>getComponentList(self, component_type[, ...])</code>	Returns a list of atoms or bonds matching the descriptor
<code>getIndexedAtoms(self)</code>	returns the list of Atom objects with an index
<code>getIndexedBonds(self)</code>	Returns a list of Bond objects that connect two indexed atoms
<code>getNeighbors(self, atom)</code>	Returns atoms that are bound to the given atom in the form of a list of Atom objects
<code>getType(self)</code>	Uses number of indexed atoms and bond connectivity to determine the type of chemical environment
<code>getUnindexedAtoms(self)</code>	returns a list of Atom objects that are not indexed
<code>getUnindexedBonds(self)</code>	Returns a list of Bond objects that connect
<code>getValence(self, atom)</code>	Returns the valence (number of neighboring atoms) around the given atom
<code>isAlpha(self, component)</code>	Takes an atom or bond and returns True if it is alpha to an indexed atom
<code>isBeta(self, component)</code>	Takes an atom or bond and returns True if it is beta to an indexed atom
<code>isIndexed(self, component)</code>	returns True if the atom or bond is indexed
<code>isUnindexed(self, component)</code>	returns True if the atom or bond is not indexed
<code>isValid(self[, smirks])</code>	Returns if the environment is valid, that is if it creates a parseable SMIRKS string.
<code>removeAtom(self, atom[, onlyEmpty])</code>	Remove the specified atom from the chemical environment.
<code>selectAtom(self[, descriptor])</code>	Select a random atom fitting the descriptor.
<code>selectBond(self[, descriptor])</code>	Select a random bond fitting the descriptor.
<code>validate(smirks[, ensure_valence_type, toolkit])</code>	Validate the provided SMIRKS string is valid, and if requested, tags atoms appropriate to the specified valence type.

**class Atom**(ORtypes=None, ANDtypes=None, index=None, ring=None)

Atom representation, which may have some ORtypes and ANDtypes properties.

#### Attributes

**ORtypes** [list of tuples in the form (base, [list of decorators])] where bases and decorators are both strings The descriptor types that will be combined with logical OR

**ANDtypes** [list of string] The descriptor types that will be combined with logical AND

#### Methods

<code>addANDtype(self, ANDtype)</code>	Adds ANDtype to the set for this atom.
<code>addORtype(self, ORbase, ORdecorators)</code>	Adds ORtype to the set for this atom.
<code>asSMARTS(self)</code>	Return the atom representation as SMARTS.

Continued on next page



Table 39 – continued from previous page

<code>asSMIRKS(self)</code>	Return the atom representation as SMIRKS.
<code>getANDtypes(self)</code>	returns a copy of the list of ANDtypes for this atom
<code>getORtypes(self)</code>	returns a copy of the dictionary of ORtypes for this atom
<code>setANDtypes(self, newANDtypes)</code>	sets new ANDtypes for this atom
<code>setORtypes(self, newORtypes)</code>	sets new ORtypes for this atom

**asSMARTS(*self*)**

Return the atom representation as SMARTS.

**Returns**

**smarts** [str]

The SMARTS string for this atom

**asSMIRKS(*self*)**

Return the atom representation as SMIRKS.

**Returns**

**smirks** [str]

The SMIRKS string for this atom

**addORtype(*self*, *ORbase*, *ORdecorators*)**

Adds ORtype to the set for this atom.

**Parameters**

**ORbase:** string, such as '#6'

**ORdecorators:** list of strings, such as ['X4','+0']

**addANDtype(*self*, *ANDtype*)**

Adds ANDtype to the set for this atom.

**Parameters**

**ANDtype:** string added to the list of ANDtypes for this atom

**getORtypes(*self*)**

returns a copy of the dictionary of ORtypes for this atom

**setORtypes(*self*, *newORtypes*)**

sets new ORtypes for this atom

**Parameters**

**newORtypes:** list of tuples in the form (base, [ORdecorators]) for example:  
 ('#6', ['X4','HO','+0']) -> '#6X4HO+0'

**getANDtypes(*self*)**

returns a copy of the list of ANDtypes for this atom

**setANDtypes(*self*, *newANDtypes*)**

sets new ANDtypes for this atom

**Parameters**

**newANDtypes:** list of strings strings that will be AND'd together in a SMARTS

**class Bond(*ORtypes*=None, *ANDtypes*=None)**

Bond representation, which may have ORtype and ANDtype descriptors.

**Attributes**

**ORtypes** [list of tuples of ORbases and ORdecorators] in form (base: [list of decorators]) The ORtype types that will be combined with logical OR

**ANDtypes** [list of string] The ANDtypes that will be combined with logical AND

## Methods

<code>addANDtype(self, ANDtype)</code>	Adds ANDtype to the set for this atom.
<code>addORtype(self, ORbase, ORdecorators)</code>	Adds ORtype to the set for this atom.
<code>asSMARTS(self)</code>	Return the atom representation as SMARTS.
<code>asSMIRKS(self)</code>	

### Returns

<code>getANDtypes(self)</code>	returns a copy of the list of ANDtypes for this atom
<code>getORtypes(self)</code>	returns a copy of the dictionary of ORtypes for this atom
<code>getOrder(self)</code>	Returns a float for the order of this bond for multiple ORtypes or ~ it returns the minimum possible order the intended application is for checking valence around a given atom
<code>setANDtypes(self, newANDtypes)</code>	sets new ANDtypes for this atom
<code>setORtypes(self, newORtypes)</code>	sets new ORtypes for this atom

**asSMARTS(self)**

Return the atom representation as SMARTS.

#### Returns

**smarts** [str] The SMARTS string for just this atom

**asSMIRKS(self)**

#### Returns

**smarts** [str] The SMIRKS string for just this bond

**getOrder(self)**

Returns a float for the order of this bond for multiple ORtypes or ~ it returns the minimum possible order the intended application is for checking valence around a given atom

**addANDtype(self, ANDtype)**

Adds ANDtype to the set for this atom.

#### Parameters

**ANDtype:** string added to the list of ANDtypes for this atom

**addORtype(self, ORbase, ORdecorators)**

Adds ORtype to the set for this atom.

#### Parameters

**ORbase:** string, such as '#6'

**ORdecorators:** list of strings, such as ['X4','+0']

**getANDtypes(self)**

returns a copy of the list of ANDtypes for this atom

**getORtypes(self)**

returns a copy of the dictionary of ORtypes for this atom

**setANDtypes(self, newANDtypes)**

sets new ANDtypes for this atom

#### Parameters

**newANDtypes:** list of strings strings that will be AND'd together in a SMARTS

**setORtypes(self, newORtypes)**

sets new ORtypes for this atom

#### Parameters

**newORtypes:** list of tuples in the form (base, [ORdecorators]) for example:  
 (`#6`, [`'X4'`, `'HO'`, `'O'`]) -> `#6X4HO+O`

**static validate**(*smirks*, *ensure\_valence\_type*=None, *toolkit*='openeye')

Validate the provided SMIRKS string is valid, and if requested, tags atoms appropriate to the specified valence type.

#### Parameters

**smirks** [str] The SMIRKS expression to validate

**ensure\_valence\_type** [str, optional, default=None] If specified, ensure the tagged atoms are appropriate to the specified valence type

This method will raise a :class:'SMIRKSParsingError' if the provided SMIRKS string is not valid.

**isValid**(*self*, *smirks*=None)

Returns if the environment is valid, that is if it creates a parseable SMIRKS string.

**asSMIRKS**(*self*, *smarts*=False)

Returns a SMIRKS representation of the chemical environment

#### Parameters

**smarts:** optional, boolean if True, returns a SMARTS instead of SMIRKS without index labels

**selectAtom**(*self*, *descriptor*=None)

Select a random atom fitting the descriptor.

#### Parameters

**descriptor:** optional, None None - returns any atom with equal probability int - will return an atom with that index 'Indexed' - returns a random indexed atom 'Unindexed' - returns a random unindexed atom 'Alpha' - returns a random alpha atom 'Beta' - returns a random beta atom

#### Returns

a single Atom object fitting the description  
 or None if no such atom exists

**getComponentList**(*self*, *component\_type*, *descriptor*=None)

Returns a list of atoms or bonds matching the descriptor

#### Parameters

**component\_type:** string: 'atom' or 'bond'

**descriptor:** string, optional 'all', 'Indexed', 'Unindexed', 'Alpha', 'Beta'

**selectBond**(*self*, *descriptor*=None)

Select a random bond fitting the descriptor.

#### Parameters

**descriptor:** optional, None None - returns any bond with equal probability int - will return an bond with that index 'Indexed' - returns a random indexed bond 'Unindexed' - returns a random unindexed bond 'Alpha' - returns a random alpha bond 'Beta' - returns a random beta bond

#### Returns

a single Bond object fitting the description

or None if no such atom exists

**addAtom**(*self*, *bondToAtom*, *bondORtypes=None*, *bondANDtypes=None*, *newORtypes=None*, *newANDtypes=None*, *newAtomIndex=None*, *newAtomRing=None*, *beyondBeta=False*)  
Add an atom to the specified target atom.

#### Parameters

**bondToAtom:** atom object, required atom the new atom will be bound to

**bondORtypes:** list of tuples, optional strings that will be used for the ORtypes for the new bond

**bondANDtypes:** list of strings, optional strings that will be used for the ANDtypes for the new bond

**newORtypes:** list of strings, optional strings that will be used for the ORtypes for the new atom

**newANDtypes:** list of strings, optional strings that will be used for the ANDtypes for the new atom

**newAtomIndex:** int, optional integer label that could be used to index the atom in a SMIRKS string

**beyondBeta:** boolean, optional if True, allows bonding beyond beta position

#### Returns

**newAtom:** atom object for the newly created atom

**removeAtom**(*self*, *atom*, *onlyEmpty=False*)  
Remove the specified atom from the chemical environment. if the atom is not indexed for the SMIRKS string or used to connect two other atoms.

#### Parameters

**atom:** atom object, required atom to be removed if it meets the conditions.

**onlyEmpty:** boolean, optional True only an atom with no ANDtypes and 1 ORtype can be removed

#### Returns

Boolean True: atom was removed, False: atom was not removed

**getAtoms**(*self*)

#### Returns

list of atoms in the environment

**getBonds**(*self*, *atom=None*)

#### Parameters

**atom:** Atom object, optional, returns bonds connected to atom

returns all bonds in fragment if atom is None

#### Returns

a complete list of bonds in the fragment

**getBond**(*self*, *atom1*, *atom2*)  
Get bond between two atoms

#### Parameters

**atom1 and atom2: atom objects**

**Returns**

**bond object between the atoms or None if no bond there**

**getIndexedAtoms(*self*)**

returns the list of Atom objects with an index

**getUnindexedAtoms(*self*)**

returns a list of Atom objects that are not indexed

**getAlphaAtoms(*self*)**

**Returns a list of atoms alpha to any indexed atom** that are not also indexed

**getBetaAtoms(*self*)**

**Returns a list of atoms beta to any indexed atom** that are not alpha or indexed atoms

**getIndexedBonds(*self*)**

Returns a list of Bond objects that connect two indexed atoms

**getUnindexedBonds(*self*)**

**Returns a list of Bond objects that connect** an indexed atom to an unindexed atom two unindexed atoms

**getAlphaBonds(*self*)**

**Returns a list of Bond objects that connect** an indexed atom to alpha atoms

**getBetaBonds(*self*)**

**Returns a list of Bond objects that connect** alpha atoms to beta atoms

**isAlpha(*self*, *component*)**

Takes an atom or bond and returns True if it is alpha to an indexed atom

**isUnindexed(*self*, *component*)**

returns True if the atom or bond is not indexed

**isIndexed(*self*, *component*)**

returns True if the atom or bond is indexed

**isBeta(*self*, *component*)**

Takes an atom or bond and returns True if it is beta to an indexed atom

**getType(*self*)**

Uses number of indexed atoms and bond connectivity to determine the type of chemical environment

**Returns**

**chemical environment type:** 'Atom', 'Bond', 'Angle', 'ProperTorsion', 'ImproperTorsion' None if number of indexed atoms is 0 or > 4

**getNeighbors(*self*, *atom*)**

Returns atoms that are bound to the given atom in the form of a list of Atom objects

**getValence(*self*, *atom*)**

Returns the valence (number of neighboring atoms) around the given atom

**getBondOrder(*self*, *atom*)**

Returns minimum bond order around a given atom 0 if atom has no neighbors aromatic bonds count as 1.5 any bond counts as 1.0

## 2.2.2 Forcefield typing engines

Engines for applying parameters to chemical systems

### 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 system creation using a ForceField, see `examples/SMIRNOFF_simulation`.

---

ForceField	A factory that assigns SMIRNOFF parameters to a molecular system
------------	--

---

#### `openforcefield.typing.engines.smirnoff.forcefield.ForceField`

```
class openforcefield.typing.engines.smirnoff.forcefield.ForceField(*sources,
                                                                    parameter_handler_classes=None,
                                                                    parameter_io_handler_classes=None,
                                                                    disable_version_check=False,
                                                                    allow_cosmetic_attributes=False)
```

A factory that assigns SMIRNOFF parameters to a molecular system

ForceField is a factory that constructs an OpenMM `simtk.openmm.System` object from a `openforcefield.topology.Topology` object defining a (bio)molecular system containing one or more molecules.

When a ForceField object is created from one or more specified SMIRNOFF serialized representations, all ParameterHandler subclasses currently imported are identified and registered to handle different sections of the SMIRNOFF force field definition file(s).

All ParameterIOHandler subclasses currently imported are identified and registered to handle different serialization formats (such as XML).

The force field definition is processed by these handlers to populate the ForceField object model data structures that can easily be manipulated via the API:

Processing a Topology object defining a chemical system will then call all `:class'ParameterHandler'` objects in an order guaranteed to satisfy the declared processing order constraints of each `:class'ParameterHandler'`.

#### Examples

Create a new ForceField containing the `smirnoff99Frosst` parameter set:

```
>>> from openforcefield.typing.engines.smirnoff import ForceField
>>> forcefield = ForceField('test_forcefields/smirnoff99Frosst.offxml')
```

Create an OpenMM system from a `openforcefield.topology.Topology` object:

```
>>> from openforcefield.topology import Molecule, Topology
>>> ethanol = Molecule.from_smiles('CCO')
>>> topology = Topology.from_molecules(molecules=[ethanol])
>>> system = forcefield.create_openmm_system(topology)
```

Modify the long-range electrostatics method:

```
>>> forcefield.get_parameter_handler('Electrostatics').method = 'PME'
```

Inspect the first few vdW parameters:

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

Retrieve the vdW parameters by SMIRKS string and manipulate it:

```
>>> parameter = forcefield.get_parameter_handler('vdW').parameters['[#1:1]-[#7]']
>>> parameter.rmin_half += 0.1 * unit.angstroms
>>> parameter.epsilon *= 1.02
```

Make a child vdW type more specific (checking modified SMIRKS for validity):

```
>>> forcefield.get_parameter_handler('vdW').parameters[-1].smirks += '~[#53]'
```

**Warning:** While we check whether the modified SMIRKS is still valid and has the appropriate valence type, we currently don't check whether the typing remains hierarchical, which could result in some types no longer being assignable because more general types now come *below* them and preferentially match.

Delete a parameter:

```
>>> del forcefield.get_parameter_handler('vdW').parameters['[#1:1]-[#6X4]']
```

Insert a parameter at a specific point in the parameter tree:

```
>>> from openforcefield.typing.engines.smirnoff import vdWHandler
>>> new_parameter = vdWHandler.vdWType(smirks='[*:1]', epsilon=0.0157*unit.kilocalories_per_mole,
↳ rmin_half=0.6000*unit.angstroms)
>>> forcefield.get_parameter_handler('vdW').parameters.insert(0, new_parameter)
```

**Warning:** We currently don't check whether removing a parameter could accidentally remove the root type, so it's possible to no longer type all molecules this way.

## Attributes

**parameters** [dict of str] `parameters[tagname]` is the instantiated `ParameterHandler` class that handles parameters associated with the force tagname. This is the primary means of retrieving and modifying parameters, such as `parameters['vdW'][0].sigma *= 1.1`

**parameter\_object\_handlers** [dict of str] Registered list of `ParameterHandler` classes that will handle different forcefield tags to create the parameter object model. `parameter_object_handlers[tagname]` is the `ParameterHandler` that will be instantiated to process the force field definition section tagname. `ParameterHandler` classes are registered when the `ForceField` object is created, but can be manipulated afterwards.

**parameter\_io\_handlers** [dict of str] Registered list of `ParameterIOHandler` classes that will handle serializing/deserializing the parameter object model to string or file representations, such as XML. `parameter_io_handlers[iotype]` is the `ParameterHandler` that will be instantiated to process the serialization scheme iotype. `ParameterIOHandler` classes are registered when the `ForceField` object is created, but can be manipulated afterwards.

## Methods

<code>create_openmm_system(self, topology, \*\*kwargs)</code>	Create an OpenMM System representing the interactions for the specified Topology with the current force field
<code>create_parmed_structure(self, topology, ...)</code>	Create a ParmEd Structure object representing the interactions for the specified Topology with the current force field
<code>get_parameter_handler(self, tagname[, ...])</code>	Retrieve the parameter handlers associated with the provided tagname.
<code>get_parameter_io_handler(self, io_format)</code>	Retrieve the parameter handlers associated with the provided tagname.
<code>label_molecules(self, topology)</code>	Return labels for a list of molecules corresponding to parameters from this force field.
<code>parse_smirnoff_from_source(self, source)</code>	Reads a SMIRNOFF data structure from a source, which can be one of many types.
<code>parse_sources(self, sources[, ...])</code>	Parse a SMIRNOFF force field definition.
<code>register_parameter_handler(self, ...)</code>	Register a new <code>ParameterHandler</code> for a specific tag, making it available for lookup in the ForceField.
<code>register_parameter_io_handler(self, ...)</code>	Register a new <code>ParameterIOHandler</code> , making it available for lookup in the ForceField.
<code>to_file(self, filename[, io_format, ...])</code>	Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.
<code>to_string(self[, io_format, ...])</code>	Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.

**`__init__`**(self, \*sources, parameter\_handler\_classes=None, parameter\_io\_handler\_classes=None, disable\_version\_check=False, allow\_cosmetic\_attributes=False)  
Create a new `ForceField` object from one or more SMIRNOFF parameter definition files.

## Parameters

**sources** [string or file-like object or open file handle or URL (or iterable of these)]  
A list of files defining the SMIRNOFF force field to be loaded. Currently, only the `SMIRNOFF XML format` is supported. Each entry may be an absolute file path, a path relative to the current working directory, a path relative to this module's data



subdirectory (for built in force fields), or an open file-like object with a `read()` method from which the forcefield 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.

**parameter\_handler\_classes** [iterable of `ParameterHandler` classes, optional, default=None] If not None, the specified set of `ParameterHandler` classes will be instantiated to create the parameter object model. By default, all imported subclasses of `ParameterHandler` are automatically registered.

**parameter\_io\_handler\_classes** [iterable of `ParameterIOHandler` classes] If not None, the specified set of `ParameterIOHandler` classes will be used to parse/generate serialized parameter sets. By default, all imported subclasses of `ParameterIOHandler` are automatically registered.

**disable\_version\_check** [bool, optional, default=False] If True, will disable checks against the current highest supported forcefield version. This option is primarily intended for forcefield development.

**allow\_cosmetic\_attributes** [bool, optional. Default = False] Whether to retain non-spec kwargs from data sources.

## Examples

Load one SMIRNOFF parameter set in XML format (searching the package data directory by default, which includes some standard parameter sets):

```
>>> forcefield = ForceField('test_forcefields/smirnoff99Frosst.offxml')
```

Load multiple SMIRNOFF parameter sets:

```
forcefield = ForceField('test_forcefields/smirnoff99Frosst.offxml', 'test_forcefields/tip3p.offxml')
```

Load a parameter set from a string:

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

## Methods

<code>__init__(self, \*sources[, ...])</code>	Create a new <code>ForceField</code> object from one or more SMIRNOFF parameter definition files.
<code>create_openmm_system(self, topology, \*\*kwargs)</code>	Create an OpenMM System representing the interactions for the specified Topology with the current force field
<code>create_parmed_structure(self, topology, ...)</code>	Create a ParmEd Structure object representing the interactions for the specified Topology with the current force field
<code>get_parameter_handler(self, tagname[, ...])</code>	Retrieve the parameter handlers associated with the provided tagname.
<code>get_parameter_io_handler(self, io_format)</code>	Retrieve the parameter handlers associated with the provided tagname.

Continued on next page

Table 43 – continued from previous page

<code>label_molecules(self, topology)</code>	Return labels for a list of molecules corresponding to parameters from this force field.
<code>parse_smirnoff_from_source(self, source)</code>	Reads a SMIRNOFF data structure from a source, which can be one of many types.
<code>parse_sources(self, sources[, ...])</code>	Parse a SMIRNOFF force field definition.
<code>register_parameter_handler(self, ...)</code>	Register a new ParameterHandler for a specific tag, making it available for lookup in the ForceField.
<code>register_parameter_io_handler(self, ...)</code>	Register a new ParameterIOHandler, making it available for lookup in the ForceField.
<code>to_file(self, filename[, io_format, ...])</code>	Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.
<code>to_string(self[, io_format, ...])</code>	Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.

### Attributes

<code>author</code>	Returns the author data for this ForceField object.
<code>date</code>	Returns the date data for this ForceField object.

#### property `author`

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

#### Returns

**author** [str] The author data for this forcefield.

#### property `date`

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

#### Returns

**date** [str] The date data for this forcefield.

#### `register_parameter_handler(self, parameter_handler)`

Register a new ParameterHandler for a specific tag, making it available for lookup in the ForceField.

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

### Parameters

**parameter\_handler** [A ParameterHandler object] The ParameterHandler to register. The TAGNAME attribute of this object will be used as the key for registration.

#### `register_parameter_io_handler(self, parameter_io_handler)`

Register a new ParameterIOHandler, making it available for lookup in the ForceField.

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

### Parameters

**parameter\_io\_handler** [A ParameterIOHandler object] The ParameterIOHandler to register. The FORMAT attribute of this object will be used to associate it to a file format/suffix.

**get\_parameter\_handler**(*self*, *tagname*, *handler\_kwargs=None*, *allow\_cosmetic\_attributes=False*)  
Retrieve the parameter handlers associated with the provided tagname.

If the parameter handler has not yet been instantiated, it will be created and returned. If a parameter handler object already exists, it will be checked for compatibility and an Exception raised if it is incompatible with the provided kwargs. If compatible, the existing ParameterHandler will be returned.

### Parameters

**tagname** [str] The name of the parameter to be handled.

**handler\_kwargs** [dict, optional. Default = None] Dict to be passed to the handler for construction or checking compatibility. If this is None and no existing ParameterHandler exists for the desired tag, a handler will be initialized with all default values. If this is None and a handler for the desired tag exists, the existing ParameterHandler will be returned.

**allow\_cosmetic\_attributes** [bool, optional. Default = False] Whether to permit non-spec kwargs in smirnoff\_data.

### Returns

**handler** [An openforcefield.engines.typing.smirnoff.ParameterHandler]

### Raises

**KeyError** if there is no ParameterHandler for the given tagname

**get\_parameter\_io\_handler**(*self*, *io\_format*)

Retrieve the parameter handlers associated with the provided tagname. If the parameter IO handler has not yet been instantiated, it will be created.

### Parameters

**io\_format** [str] The name of the io format to be handled.

### Returns

**io\_handler** [An openforcefield.engines.typing.smirnoff.ParameterIOHandler]

### Raises

**KeyError** if there is no ParameterIOHandler for the given tagname

**parse\_sources**(*self*, *sources*, *allow\_cosmetic\_attributes=True*)

Parse a SMIRNOFF force field definition.

### Parameters

**sources** [string or file-like object or open file handle or URL (or iterable of these)]  
A list of files defining the SMIRNOFF force field to be loaded. Currently, only the [SMIRNOFF XML format](#) is supported. Each entry may be an absolute file path, a path relative to the current working directory, a path relative to this module's data

subdirectory (for built in force fields), or an open file-like object with a `read()` method from which the forcefield XML data can be loaded. If multiple files are specified, any top-level tags that are repeated will be merged if they are compatible, with files appearing later in the sequence resulting in parameters that have higher precedence. Support for multiple files is primarily intended to allow solvent parameters to be specified by listing them last in the sequence.

**allow\_cosmetic\_attributes** [bool, optional. Default = False] Whether to permit non-spec kwargs present in the source.

**.. notes ::**

- New SMIRNOFF sections are handled independently, as if they were specified in the same file.
- If a SMIRNOFF section that has already been read appears again, its definitions are appended to the end of the previously-read definitions if the sections are configured with compatible attributes; otherwise, an `IncompatibleTagException` is raised.

**parse\_smirnoff\_from\_source**(*self*, *source*)

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

#### Parameters

**source** [str or bytes] sources : string or file-like object or open file handle or URL (or iterable of these) A list of files defining the SMIRNOFF force field to be loaded. Currently, only [the SMIRNOFF XML format](#) is supported. Each entry may be an absolute file path, a path relative to the current working directory, a path relative to this module's data subdirectory (for built in force fields), or an open file-like object with a `read()` method from which the forcefield XML data can be loaded.

#### Returns

**smirnoff\_data** [OrderedDict] A representation of a SMIRNOFF-format data structure. Begins at top-level 'SMIRNOFF' key.

**to\_string**(*self*, *io\_format*='XML', *discard\_cosmetic\_attributes*=False)

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

#### Parameters

**io\_format** [str or ParameterIOHandler, optional. Default='XML'] The serialization format to write to

**discard\_cosmetic\_attributes** [bool, default=False] Whether to discard any non-spec attributes stored in the ForceField.

#### Returns

**forcefield\_string** [str] The string representation of the serialized forcefield

**to\_file**(*self*, *filename*, *io\_format*=None, *discard\_cosmetic\_attributes*=False)

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

#### Parameters

**filename** [str] The filename to write to

**io\_format** [str or ParameterIOHandler, optional. Default=None] The serialization format to write out. If None, will attempt to be inferred from the filename.

**discard\_cosmetic\_attributes** [bool, default=False] Whether to discard any non-spec attributes stored in the ForceField.

#### Returns

**forcefield\_string** [str] The string representation of the serialized forcefield

**create\_openmm\_system**(*self*, *topology*, *\*\*kwargs*)

Create an OpenMM System representing the interactions for the specified Topology with the current force field

#### Parameters

**topology** [openforcefield.topology.Topology] The Topology corresponding to the system to be parameterized

**charge\_from\_molecules** [List[openforcefield.molecule.Molecule], optional] If specified, partial charges will be taken from the given molecules instead of being determined by the force field.

#### Returns

**system** [simtk.openmm.System] The newly created OpenMM System corresponding to the specified topology

**create\_parmed\_structure**(*self*, *topology*, *positions*, *\*\*kwargs*)

Create a ParmEd Structure object representing the interactions for the specified Topology with the current force field

This method creates a [ParmEd](#) Structure object containing a topology, positions, and parameters.

#### Parameters

**topology** [openforcefield.topology.Topology] The Topology corresponding to the System object to be created.

**positions** [simtk.unit.Quantity of dimension (natoms,3) with units compatible with angstroms] The positions corresponding to the System object to be created

#### Returns

**structure** [parmed.Structure] The newly created parmed.Structure object

**label\_molecules**(*self*, *topology*)

Return labels for a list of molecules corresponding to parameters from this force field. For each molecule, a dictionary of force types is returned, and for each force type, each force term is provided with the atoms involved, the parameter id assigned, and the corresponding SMIRKS.

#### Parameters

**topology** [openforcefield.topology.Topology] A Topology object containing one or more unique molecules to be labeled

#### Returns

**molecule\_labels** [list] List of labels for unique molecules. Each entry in the list corresponds to one unique molecule in the Topology and is a dictionary keyed by force type, i.e., `molecule_labels[0]['HarmonicBondForce']` gives details for the harmonic bond parameters for the first molecule. Each element is a list of the form: [ ( [ atom1, ..., atomN], parameter\_id, SMIRKS), ... ].

## 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`.

<code>ParameterType</code>	Base class for SMIRNOFF parameter types.
<code>BondHandler.BondType</code>	A SMIRNOFF bond type
<code>AngleHandler.AngleType</code>	A SMIRNOFF angle type.
<code>ProperTorsionHandler.ProperTorsionType</code>	A SMIRNOFF torsion type for proper torsions.
<code>ImproperTorsionHandler.ImproperTorsionType</code>	A SMIRNOFF torsion type for improper torsions.
<code>vdWHandler.vdWType</code>	A SMIRNOFF vdWForce type.
<code>GBSAHandler.GBSAType</code>	A SMIRNOFF GBSA type.

## `openforcefield.typing.engines.smirnoff.parameters.ParameterType`

```
class openforcefield.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.

See also:

`ParameterAttribute`

`IndexedParameterAttribute`

## Examples

This class allows to define new parameter types by just listing its attributes. In the example below, `_VALENCE_TYPE` AND `_ELEMENT_NAME` are used for the validation of the SMIRKS pattern associated to the parameter and the automatic serialization/deserialization into a dict.

```
>>> class MyBondParameter(ParameterType):
...     _VALENCE_TYPE = 'Bond'
...     _ELEMENT_NAME = 'Bond'
...     length = ParameterAttribute(unit=unit.angstrom)
...     k = ParameterAttribute(unit=unit.kilocalorie_per_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]',
```

(continues on next page)

(continued from previous page)

```

...     length='1.01 * angstrom',
...     k=5 * unit.kilocalorie_per_mole / unit.angstrom**2
... )
>>> my_par.length
Quantity(value=1.01, unit=angstrom)
>>> my_par.k = 3.0 * unit.gram
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: k=3.0 g should have units of kilocalorie/
↳(angstrom**2*mole)

```

Each attribute can be made optional by specifying a default value, and you can attach a converter function by passing a callable as an argument or through the decorator syntax.

```

>>> class MyParameterType(ParameterType):
...     _VALENCE_TYPE = 'Atom'
...     _ELEMENT_NAME = 'Atom'
...
...     attr_optional = ParameterAttribute(default=2)
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to floats
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameterType(smirks='[*:1]', attr_all_to_float='3.0', attr_int_to_float=1)
>>> my_par.attr_optional
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 “ParameterAttribute”s, but the validation/conversion is performed for each indexed attribute.

```

>>> class MyTorsionType(ParameterType):
...     _VALENCE_TYPE = 'ProperTorsion'
...     _ELEMENT_NAME = 'Proper'
...     periodicity = IndexedParameterAttribute(converter=int)

```

(continues on next page)

(continued from previous page)

```

...     k = IndexedParameterAttribute(unit=unit.kilocalorie_per_mole)
...
>>> my_par = MyTorsionType(
...     smirks='[*:1]-[*:2]-[*:3]-[*:4]',
...     periodicity1=2,
...     k1=5 * unit.kilocalorie_per_mole,
...     periodicity2='3',
...     k2=6 * unit.kilocalorie_per_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]

```

### Attributes

**smirks** [str] The SMIRKS pattern that this parameter matches.

**id** [str or None] An optional identifier for the parameter.

**parent\_id** [str or None] Optionally, the identifier of the parameter of which this parameter is a specialization.

### Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

**\_\_init\_\_**(self, smirks, allow\_cosmetic\_attributes=False, \*\*kwargs)  
Create a ParameterType.

### Parameters

**smirks** [str] The SMIRKS match for the provided parameter type.

**allow\_cosmetic\_attributes** [bool optional. Default = False] Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

### Methods

<code>__init__(self, smirks[, ...])</code>	Create a ParameterType.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.



## Attributes

<code>id</code>	A descriptor for ParameterType attributes.
<code>parent_id</code>	A descriptor for ParameterType attributes.
<code>smirks</code>	A descriptor for ParameterType attributes.

**add\_cosmetic\_attribute**(*self*, *attr\_name*, *attr\_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**attr\_name** [str] Name of the cosmetic attribute to delete.

**to\_dict**(*self*, *discard\_cosmetic\_attributes=False*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if *discard\_cosmetic\_attributes* is False.

### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False] Whether to discard non-spec attributes of this object

### Returns

**smirnoff\_dict** [dict] The SMIRNOFF-compliant dict representation of this object.

## openforcefield.typing.engines.smirnoff.parameters.BondHandler.BondType

**class** openforcefield.typing.engines.smirnoff.parameters.BondHandler.**BondType**(*smirks*, *allow\_cosmetic\_attributes=False*, *\*\*kwargs*)

A SMIRNOFF bond type

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

## Attributes

**id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
...
(continues on next page)
```

(continued from previous page)

```

>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom

```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3` `>>> my_par.attr_int_to_float 3.0` `>>> my_par.attr_int_to_float = '4.0'` Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

#### k 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
```

(continues on next page)

(continued from previous page)

```

...     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`

**length** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```

>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()

```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**parent\_id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0'` Traceback (most recent call last): ... `TypeError: Cannot convert '4.0' to float`

**smirks** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.



**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

`__init__(self, smirks, allow_cosmetic_attributes=False, **kwargs)`

Create a `ParameterType`.

### Parameters

**smirks** [str] The SMIRKS match for the provided parameter type.

**allow\_cosmetic\_attributes** [bool optional. Default = False] Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

## Methods

<code>__init__(self, smirks[, ...])</code>	Create a <code>ParameterType</code> .
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

## Attributes

<code>id</code>	A descriptor for <code>ParameterType</code> attributes.
<code>k</code>	A descriptor for <code>ParameterType</code> attributes.
<code>length</code>	A descriptor for <code>ParameterType</code> attributes.
<code>parent_id</code>	A descriptor for <code>ParameterType</code> attributes.
<code>smirks</code>	A descriptor for <code>ParameterType</code> attributes.

`add_cosmetic_attribute(self, attr_name, attr_value)`

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the cosmetic attribute to delete.

**to\_dict**(*self*, *discard\_cosmetic\_attributes=False*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if *discard\_cosmetic\_attributes* is False.

#### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False] Whether to discard non-spec attributes of this object

#### Returns

**smirnoff\_dict** [dict] The SMIRNOFF-compliant dict representation of this object.

### openforcefield.typing.engines.smirnoff.parameters.AngleHandler.AngleType

**class** openforcefield.typing.engines.smirnoff.parameters.AngleHandler.**AngleType**(*smirks*, *allow\_cosmetic\_attributes=False*, *\*\*kwargs*)

A SMIRNOFF angle type.

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

#### Attributes

**angle** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
```

(continues on next page)

(continued from previous page)

```

...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

#### id 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> 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`

**k** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3` `>>> my_par.attr_int_to_float 3.0` `>>> my_par.attr_int_to_float = '4.0'` Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

**parent\_id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
```

(continues on next page)

(continued from previous page)

```

...     # This converter converts only integers to float
...     # and raise an exception for the other types.
...     if isinstance(value, int):
...         return float(value)
...     elif not isinstance(value, float):
...         raise TypeError(f"Cannot convert '{value}' to float")
...     return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**smirks** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```

>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()

```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3` `>>> my_par.attr_int_to_float 3.0` `>>> my_par.attr_int_to_float = '4.0'` `Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

`__init__(self, smirks, allow_cosmetic_attributes=False, **kwargs)`

Create a ParameterType.

### Parameters

**smirks** [str] The SMIRKS match for the provided parameter type.

**allow\_cosmetic\_attributes** [bool optional. Default = False] Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

## Methods

<code>__init__(self, smirks[, ...])</code>	Create a ParameterType.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

## Attributes

<code>angle</code>	A descriptor for ParameterType attributes.
<code>id</code>	A descriptor for ParameterType attributes.
<code>k</code>	A descriptor for ParameterType attributes.
<code>parent_id</code>	A descriptor for ParameterType attributes.
<code>smirks</code>	A descriptor for ParameterType attributes.

`add_cosmetic_attribute(self, attr_name, attr_value)`

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

`delete_cosmetic_attribute(self, attr_name)`

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the cosmetic attribute to delete.

**to\_dict**(self, discard\_cosmetic\_attributes=False)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard\_cosmetic\_attributes is False.

#### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False] Whether to discard non-spec attributes of this object

#### Returns

**smirnoff\_dict** [dict] The SMIRNOFF-compliant dict representation of this object.

### openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler.ProperTorsionType

**class** openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler.**ProperTorsionType**(smirks, al-  
low\_cosmetic\_at  
\*\*kwargs)

A SMIRNOFF torsion type for proper torsions.

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

#### Attributes

**id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0'` Traceback (most recent call last): ... `TypeError: Cannot convert '4.0' to float`

**idivf** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [`simtk.unit.Quantity`, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into `Quantity` objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with `ParameterAttribute`, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**k** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [`simtk.unit.Quantity`, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into `Quantity` objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with `ParameterAttribute`, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```



**parent\_id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**periodicity** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with ParameterAttribute, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**phase** 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  $k_1$ ,  $k_2$ , ..., 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
```

(continues on next page)

(continued from previous page)

```
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with ParameterAttribute, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**smirks** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

`__init__(self, smirks, allow_cosmetic_attributes=False, **kwargs)`

Create a ParameterType.

### Parameters

**smirks** [str] The SMIRKS match for the provided parameter type.

**allow\_cosmetic\_attributes** [bool optional. Default = False] Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

## Methods

<code>__init__(self, smirks[, ...])</code>	Create a ParameterType.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

## Attributes

<code>id</code>	A descriptor for ParameterType attributes.
<code>idivf</code>	The attribute of a parameter with an unspecified number of terms.
<code>k</code>	The attribute of a parameter with an unspecified number of terms.
<code>parent_id</code>	A descriptor for ParameterType attributes.
<code>periodicity</code>	The attribute of a parameter with an unspecified number of terms.
<code>phase</code>	The attribute of a parameter with an unspecified number of terms.
<code>smirks</code>	A descriptor for ParameterType attributes.

`add_cosmetic_attribute(self, attr_name, attr_value)`

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the cosmetic attribute to delete.

**to\_dict**(*self*, *discard\_cosmetic\_attributes=False*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if *discard\_cosmetic\_attributes* is False.

#### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False] Whether to discard non-spec attributes of this object

#### Returns

**smirnoff\_dict** [dict] The SMIRNOFF-compliant dict representation of this object.

### openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler.ImproperTorsionType

**class** openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler.**ImproperTorsionType**(*smirks*, *al-*  
*low\_cosmet*  
*\*\*kwargs*)

A SMIRNOFF torsion type for improper torsions.

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

#### Attributes

**id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
```

(continues on next page)



(continued from previous page)

```

...     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`

**idivf** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```

>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True

```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with `ParameterAttribute`, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**k** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into `Quantity` objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with `ParameterAttribute`, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**parent\_id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**periodicity** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with ParameterAttribute, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**phase** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with ParameterAttribute, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**smirks** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> 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`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

`__init__(self, smirks, allow_cosmetic_attributes=False, **kwargs)`

Create a `ParameterType`.

### Parameters

**smirks** [str] The SMIRKS match for the provided parameter type.

**allow\_cosmetic\_attributes** [bool optional. Default = False] Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

## Methods

<code>__init__(self, smirks[, ...])</code>	Create a <code>ParameterType</code> .
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

## Attributes

<code>id</code>	A descriptor for <code>ParameterType</code> attributes.
<code>idivf</code>	The attribute of a parameter with an unspecified number of terms.
<code>k</code>	The attribute of a parameter with an unspecified number of terms.
<code>parent_id</code>	A descriptor for <code>ParameterType</code> attributes.
<code>periodicity</code>	The attribute of a parameter with an unspecified number of terms.
<code>phase</code>	The attribute of a parameter with an unspecified number of terms.
<code>smirks</code>	A descriptor for <code>ParameterType</code> attributes.



**add\_cosmetic\_attribute**(*self*, *attr\_name*, *attr\_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the cosmetic attribute to delete.

**to\_dict**(*self*, *discard\_cosmetic\_attributes=False*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if *discard\_cosmetic\_attributes* is False.

#### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False] Whether to discard non-spec attributes of this object

#### Returns

**smirnoff\_dict** [dict] The SMIRNOFF-compliant dict representation of this object.

### openforcefield.typing.engines.smirnoff.parameters.vdWHandler.vdWType

**class** openforcefield.typing.engines.smirnoff.parameters.vdWHandler.vdWType(\*\*kwargs)  
A SMIRNOFF vdWForce type.

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

#### Attributes

**epsilon** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

#### id 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> 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`

**parent\_id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3` `>>> my_par.attr_int_to_float 3.0` `>>> my_par.attr_int_to_float = '4.0'` Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

**rmin\_half** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
```

(continues on next page)

(continued from previous page)

```

...     # This converter converts only integers to float
...     # and raise an exception for the other types.
...     if isinstance(value, int):
...         return float(value)
...     elif not isinstance(value, float):
...         raise TypeError(f"Cannot convert '{value}' to float")
...     return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**sigma** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```

>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()

```



Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3` `>>> my_par.attr_int_to_float 3.0` `>>> my_par.attr_int_to_float = '4.0'` `Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**smirks** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3` `>>> my_par.attr_int_to_float 3.0` `>>> my_par.attr_int_to_float = '4.0'` Traceback (most recent call last): ... `TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

`__init__(self, **kwargs)`  
Create a `ParameterType`.

## Parameters

**smirks** [str] The SMIRKS match for the provided parameter type.

**allow\_cosmetic\_attributes** [bool optional. Default = False] Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

## Methods

<code>__init__(self, **kwargs)</code>	Create a <code>ParameterType</code> .
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

## Attributes

<code>epsilon</code>	A descriptor for <code>ParameterType</code> attributes.
<code>id</code>	A descriptor for <code>ParameterType</code> attributes.
<code>parent_id</code>	A descriptor for <code>ParameterType</code> attributes.
<code>rmin_half</code>	A descriptor for <code>ParameterType</code> attributes.
<code>sigma</code>	A descriptor for <code>ParameterType</code> attributes.
<code>smirks</code>	A descriptor for <code>ParameterType</code> attributes.

**`add_cosmetic_attribute(self, attr_name, attr_value)`**

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**`attr_name`** [str] Name of the attribute to define for this object.

**`attr_value`** [str] The value of the attribute to define for this object.

**`delete_cosmetic_attribute(self, attr_name)`**

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**`attr_name`** [str] Name of the cosmetic attribute to delete.

**`to_dict(self, discard_cosmetic_attributes=False)`**

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

### Parameters

**`discard_cosmetic_attributes`** [bool, optional. Default = `False`] Whether to discard non-spec attributes of this object

### Returns

**smirnoff\_dict** [dict] The SMIRNOFF-compliant dict representation of this object.

## openforcefield.typing.engines.smirnoff.parameters.GBSAHandler.GBSAType

**class** openforcefield.typing.engines.smirnoff.parameters.GBSAHandler.GBSAType(\*\*kwargs)  
A SMIRNOFF GBSA type.

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

### Attributes

**id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3` `>>> my_par.attr_int_to_float 3.0` `>>> my_par.attr_int_to_float = '4.0'` Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

**parent\_id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**radius** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.



```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> 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`

**scale** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```

>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom

```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**smirks** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
```

(continues on next page)

(continued from previous page)

```

...     # This converter converts only integers to float
...     # and raise an exception for the other types.
...     if isinstance(value, int):
...         return float(value)
...     elif not isinstance(value, float):
...         raise TypeError(f"Cannot convert '{value}' to float")
...     return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

`__init__(self, **kwargs)`  
Create a ParameterType.

### Parameters

**smirks** [str] The SMIRKS match for the provided parameter type.

**allow\_cosmetic\_attributes** [bool optional. Default = False] Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

## Methods

<code>__init__(self, \**kwargs)</code>	Create a ParameterType.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

## Attributes

<code>id</code>	A descriptor for <code>ParameterType</code> attributes.
<code>parent_id</code>	A descriptor for <code>ParameterType</code> attributes.
<code>radius</code>	A descriptor for <code>ParameterType</code> attributes.
<code>scale</code>	A descriptor for <code>ParameterType</code> attributes.
<code>smirks</code>	A descriptor for <code>ParameterType</code> attributes.

**`add_cosmetic_attribute(self, attr_name, attr_value)`**

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**`attr_name`** [str] Name of the attribute to define for this object.

**`attr_value`** [str] The value of the attribute to define for this object.

**`delete_cosmetic_attribute(self, attr_name)`**

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**`attr_name`** [str] Name of the cosmetic attribute to delete.

**`to_dict(self, discard_cosmetic_attributes=False)`**

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

#### Parameters

**`discard_cosmetic_attributes`** [bool, optional. Default = `False`] Whether to discard non-spec attributes of this object

#### Returns

**`smirnoff_dict`** [dict] The SMIRNOFF-compliant dict representation of this object.

## Parameter Handlers

Each `ForceField` primarily consists of several `ParameterHandler` objects, which each contain the machinery to add one energy component to a system. During system creation, each `ParameterHandler` registered to a `ForceField` has its `assign_parameters()` function called..

<code>ParameterList</code>	Parameter list that also supports accessing items by SMARTS string.
<code>ParameterHandler</code>	Base class for parameter handlers.
<code>BondHandler</code>	Handle SMIRNOFF <Bonds> tags
<code>AngleHandler</code>	Handle SMIRNOFF <AngleForce> tags
<code>ProperTorsionHandler</code>	Handle SMIRNOFF <ProperTorsionForce> tags
<code>ImproperTorsionHandler</code>	Handle SMIRNOFF <ImproperTorsionForce> tags
<code>vdWHandler</code>	Handle SMIRNOFF <vdW> tags
<code>ElectrostaticsHandler</code>	Handles SMIRNOFF <Electrostatics> tags.
<code>ToolkitAM1BCCHandler</code>	Handle SMIRNOFF <ToolkitAM1BCC> tags
<code>GBSAHandler</code>	Handle SMIRNOFF <GBSA> tags

### openforcefield.typing.engines.smirnoff.parameters.ParameterList

**class** openforcefield.typing.engines.smirnoff.parameters.**ParameterList**(*input\_parameter\_list=None*)  
Parameter list that also supports accessing items by SMARTS string.

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

#### Methods

<code>append(self, parameter)</code>	Add a <code>ParameterType</code> object to the end of the <code>ParameterList</code>
<code>clear(self, /)</code>	Remove all items from list.
<code>copy(self, /)</code>	Return a shallow copy of the list.
<code>count(self, value, /)</code>	Return number of occurrences of value.
<code>extend(self, other)</code>	Add a <code>ParameterList</code> object to the end of the <code>ParameterList</code>
<code>index(self, item)</code>	Get the numerical index of a <code>ParameterType</code> object or SMIRKS in this <code>ParameterList</code> .
<code>insert(self, index, parameter)</code>	Add a <code>ParameterType</code> object as if this were a list
<code>pop(self[, index])</code>	Remove and return item at index (default last).
<code>remove(self, value, /)</code>	Remove first occurrence of value.
<code>reverse(self, /)</code>	Reverse <i>IN PLACE</i> .
<code>sort(self, /, \*[, key, reverse])</code>	Stable sort <i>IN PLACE</i> .
<code>to_list(self[, discard_cosmetic_attributes])</code>	Render this <code>ParameterList</code> to a normal list, serializing each <code>ParameterType</code> object in it to dict.

**\_\_init\_\_(self, input\_parameter\_list=None)**  
Initialize a new `ParameterList`, optionally providing a list of `ParameterType` objects to initially populate it.

#### Parameters

**input\_parameter\_list:** `list[ParameterType]`, `default=None` A pre-existing list of `ParameterType`-based objects. If `None`, this `ParameterList` will be initialized empty.

## Methods

<code>__init__(self[, input_parameter_list])</code>	Initialize a new <code>ParameterList</code> , optionally providing a list of <code>ParameterType</code> objects to initially populate it.
<code>append(self, parameter)</code>	Add a <code>ParameterType</code> object to the end of the <code>ParameterList</code>
<code>clear(self, /)</code>	Remove all items from list.
<code>copy(self, /)</code>	Return a shallow copy of the list.
<code>count(self, value, /)</code>	Return number of occurrences of value.
<code>extend(self, other)</code>	Add a <code>ParameterList</code> object to the end of the <code>ParameterList</code>
<code>index(self, item)</code>	Get the numerical index of a <code>ParameterType</code> object or SMIRKS in this <code>ParameterList</code> .
<code>insert(self, index, parameter)</code>	Add a <code>ParameterType</code> object as if this were a list
<code>pop(self[, index])</code>	Remove and return item at index (default last).
<code>remove(self, value, /)</code>	Remove first occurrence of value.
<code>reverse(self, /)</code>	Reverse <i>IN PLACE</i> .
<code>sort(self, /, \*[, key, reverse])</code>	Stable sort <i>IN PLACE</i> .
<code>to_list(self[, discard_cosmetic_attributes])</code>	Render this <code>ParameterList</code> to a normal list, serializing each <code>ParameterType</code> object in it to dict.

**`append(self, parameter)`**

Add a `ParameterType` object to the end of the `ParameterList`

### Parameters

**parameter** [a `ParameterType` object]

**`extend(self, other)`**

Add a `ParameterList` object to the end of the `ParameterList`

### Parameters

**other** [a `ParameterList`]

**`index(self, item)`**

Get the numerical index of a `ParameterType` object or SMIRKS in this `ParameterList`. Raises `ValueError` if the item is not found.

### Parameters

**item** [`ParameterType` object or str] The parameter or SMIRKS to look up in this `ParameterList`

### Returns

**index** [int] The index of the found item

**`insert(self, index, parameter)`**

Add a `ParameterType` object as if this were a list

### Parameters

**index** [int] The numerical position to insert the parameter at

**parameter** [a `ParameterType` object] The parameter to insert

**`to_list(self, discard_cosmetic_attributes=True)`**

Render this `ParameterList` to a normal list, serializing each `ParameterType` object in it to dict.



### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = True] Whether to discard non-spec attributes of each ParameterType object.

### Returns

**parameter\_list** [List[dict]] A serialized representation of a ParameterList, with each ParameterType it contains converted to dict.

**clear**(self, /)

Remove all items from list.

**copy**(self, /)

Return a shallow copy of the list.

**count**(self, value, /)

Return number of occurrences of value.

**pop**(self, index=-1, /)

Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

**remove**(self, value, /)

Remove first occurrence of value.

Raises ValueError if the value is not present.

**reverse**(self, /)

Reverse *IN PLACE*.

**sort**(self, /, \*, key=None, reverse=False)

Stable sort *IN PLACE*.

## openforcefield.typing.engines.smirnoff.parameters.ParameterHandler

```
class openforcefield.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.

### Attributes

**known\_kwargs** List of kwargs that can be parsed by the function.

**parameters** The ParameterList that holds this ParameterHandler's parameter objects

**version** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0'` Traceback (most recent call last): ... `TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, handler_kwargs)</code>	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the <code>parameter_attrs</code> argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

```

__init__(self, allow_cosmetic_attributes=False, skip_version_check=False, **kwargs)
    Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

```

## Parameters

**allow\_cosmetic\_attributes** [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

**skip\_version\_check**: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to `_MAX_SUPPORTED_SECTION_VERSION`.

**\*\*kwargs** [dict] The dict representation of the SMIRNOFF data source

## Methods

<code>__init__(self[, allow_cosmetic_attributes, ...])</code>	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, handler_kwargs)</code>	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the <code>parameter_attrs</code> argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

## Attributes

<code>known_kwargs</code>	List of kwargs that can be parsed by the function.
<code>parameters</code>	The ParameterList that holds this ParameterHandler's parameter objects
<code>version</code>	A descriptor for ParameterType attributes.

### property parameters

The ParameterList that holds this ParameterHandler's parameter objects

### property known\_kwargs

List of kwargs that can be parsed by the function.

### check\_handler\_compatibility(self, handler\_kwargs)

Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

## Parameters

**handler\_kwargs** [dict] The kwargs that would be used to construct

#### Raises

**IncompatibleParameterError** if **handler\_kwargs** are incompatible with existing parameters.

**add\_parameter**(*self*, *parameter\_kwargs*)

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

#### Parameters

**parameter\_kwargs** [dict] The kwargs to pass to the `ParameterHandler.INFO_TYPE` (a `ParameterType`) constructor

**get\_parameter**(*self*, *parameter\_attrs*)

Return the parameters in this `ParameterHandler` that match the *parameter\_attrs* argument

#### Parameters

**parameter\_attrs** [dict of {attr: value}] The attrs mapped to desired values (for example {“smirks”: “[:1]~[#16:2]=,:[#6:3]~[:4]”, “id”: “t105”} )

#### Returns

**list of `ParameterType` objects** A list of matching `ParameterType` objects

**find\_matches**(*self*, *entity*)

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

#### Parameters

**entity** [`openforcefield.topology.Topology`] Topology to search.

#### Returns

**matches** [`ValenceDict`[`Tuple`[int], `ParameterHandler.Match`]]  
 matches[particle\_indices] is the `ParameterType` object matching the tuple of particle indices in entity.

**assign\_parameters**(*self*, *topology*, *system*)

Assign parameters for the given `Topology` to the specified `System` object.

#### Parameters

**topology** [`openforcefield.topology.Topology`] The `Topology` for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [`simtk.openmm.System`] The `OpenMM System` object to add the Force (or append new parameters) to.

**postprocess\_system**(*self*, *topology*, *system*, *\*\*kwargs*)

Allow the force to perform a final post-processing pass on the `System` following parameter assignment, if needed.

#### Parameters

**topology** [`openforcefield.topology.Topology`] The `Topology` for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [`simtk.openmm.System`] The `OpenMM System` object to add the Force (or append new parameters) to.

**to\_dict**(*self*, *discard\_cosmetic\_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

**Parameters**

**discard\_cosmetic\_attributes** [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

**Returns**

**smirnoff\_data** [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

**add\_cosmetic\_attribute**(*self*, *attr\_name*, *attr\_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

**Parameters**

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

**Parameters**

**attr\_name** [str] Name of the cosmetic attribute to delete.

## openforcefield.typing.engines.smirnoff.parameters.BondHandler

**class** openforcefield.typing.engines.smirnoff.parameters.**BondHandler**(*allow\_cosmetic\_attributes=False*,  
*skip\_version\_check=False*,  
*\*\*kwargs*)

Handle SMIRNOFF <Bonds> tags

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

**Attributes**

**fractional\_bondorder\_interpolation** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**fractional\_bondorder\_method** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.



```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> 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`

**known\_kwargs** List of kwargs that can be parsed by the function.

**parameters** The `ParameterList` that holds this `ParameterHandler`'s parameter objects

**potential** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**version** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
```

(continues on next page)

(continued from previous page)

```

...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0'` Traceback (most recent call last): ... `TypeError: Cannot convert '4.0' to float`

## Methods

<code>BondType(smirks[, allow_cosmetic_attributes])</code>	A SMIRNOFF bond type
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the <code>parameter_attrs</code> argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

create\_force

**\_\_init\_\_**(self, allow\_cosmetic\_attributes=False, skip\_version\_check=False, \*\*kwargs)  
 Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

### Parameters

**allow\_cosmetic\_attributes** [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

**skip\_version\_check**: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to `_MAX_SUPPORTED_SECTION_VERSION`.

**\*\*kwargs** [dict] The dict representation of the SMIRNOFF data source

### Methods

<code>__init__(self[, allow_cosmetic_attributes, ...])</code>	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>create_force(self, system, topology, \*\*kwargs)</code>	
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the parameter_attrs argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

### Attributes

<code>fractional_bondorder_interpolation</code>	A descriptor for ParameterType attributes.
<code>fractional_bondorder_method</code>	A descriptor for ParameterType attributes.
<code>known_kwargs</code>	List of kwargs that can be parsed by the function.
<code>parameters</code>	The ParameterList that holds this ParameterHandler's parameter objects
<code>potential</code>	A descriptor for ParameterType attributes.
<code>version</code>	A descriptor for ParameterType attributes.

```
class BondType(smirks, allow_cosmetic_attributes=False, **kwargs)
    A SMIRNOFF bond type
```

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

### Attributes

**id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```

>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom

```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**k** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↪ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
```

(continues on next page)

(continued from previous page)

```
...     # This converter converts only integers to float
...     # and raise an exception for the other types.
...     if isinstance(value, int):
...         return float(value)
...     elif not isinstance(value, float):
...         raise TypeError(f"Cannot convert '{value}' to float")
...     return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**length** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**parent\_id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**smirks** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

**`add_cosmetic_attribute(self, attr_name, attr_value)`**

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**`attr_name`** [str] Name of the attribute to define for this object.

**`attr_value`** [str] The value of the attribute to define for this object.

**`delete_cosmetic_attribute(self, attr_name)`**

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**`attr_name`** [str] Name of the cosmetic attribute to delete.

**`to_dict(self, discard_cosmetic_attributes=False)`**

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

### Parameters

**`discard_cosmetic_attributes`** [bool, optional. Default = `False`] Whether to discard non-spec attributes of this object

**Returns**

**smirnoff\_dict** [dict] The SMIRNOFF-compliant dict representation of this object.

**check\_handler\_compatibility**(*self*, *other\_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

**Parameters**

**other\_handler** [a ParameterHandler object] The handler to compare to.

**Raises**

**IncompatibleParameterError** if **handler\_kwargs** are incompatible with existing parameters.

**add\_cosmetic\_attribute**(*self*, *attr\_name*, *attr\_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

**Parameters**

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

**add\_parameter**(*self*, *parameter\_kwargs*)

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

**Parameters**

**parameter\_kwargs** [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

**assign\_parameters**(*self*, *topology*, *system*)

Assign parameters for the given Topology to the specified System object.

**Parameters**

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

**Parameters**



**attr\_name** [str] Name of the cosmetic attribute to delete.

**find\_matches**(*self*, *entity*)

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

#### Parameters

**entity** [openforcefield.topology.Topology] Topology to search.

#### Returns

**matches** [ValenceDict[Tuple[int], ParameterHandler.Match]]  
 matches[particle\_indices] is the ParameterType object matching the tuple of particle indices in entity.

**get\_parameter**(*self*, *parameter\_attrs*)

Return the parameters in this ParameterHandler that match the parameter\_attrs argument

#### Parameters

**parameter\_attrs** [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[\*:1]~[#16:2]=,.[#6:3]~[:4]", "id": "t105"})

#### Returns

**list of ParameterType objects** A list of matching ParameterType objects

**property known\_kwargs**

List of kwargs that can be parsed by the function.

**property parameters**

The ParameterList that holds this ParameterHandler's parameter objects

**postprocess\_system**(*self*, *topology*, *system*, *\*\*kwargs*)

Allow the force to perform a final post-processing pass on the System following parameter assignment, if needed.

#### Parameters

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**to\_dict**(*self*, *discard\_cosmetic\_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

#### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

#### Returns

**smirnoff\_data** [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

**openforcefield.typing.engines.smirnoff.parameters.AngleHandler**

```
class openforcefield.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.

**Attributes**

**known\_kwargs** List of kwargs that can be parsed by the function.

**parameters** The ParameterList that holds this ParameterHandler's parameter objects

**potential** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:  
...     attr_required = ParameterAttribute()  
...     attr_optional = ParameterAttribute(default=2)  
...  
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional  
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**version** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0'` Traceback (most recent call last): ... `TypeError: Cannot convert '4.0' to float`

## Methods

<code>AngleType(smirks[, allow_cosmetic_attributes])</code>	A SMIRNOFF angle type.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the <code>parameter_attrs</code> argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

create_force	
--------------	--

**\_\_init\_\_**(self, allow\_cosmetic\_attributes=False, skip\_version\_check=False, \*\*kwargs)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

### Parameters

**allow\_cosmetic\_attributes** [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

**skip\_version\_check**: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to `_MAX_SUPPORTED_SECTION_VERSION`.

**\*\*kwargs** [dict] The dict representation of the SMIRNOFF data source

### Methods

<code>__init__(self[, allow_cosmetic_attributes, ...])</code>	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>create_force(self, system, topology, \*\*kwargs)</code>	
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the parameter_attrs argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

### Attributes

<code>known_kwargs</code>	List of kwargs that can be parsed by the function.
<code>parameters</code>	The ParameterList that holds this ParameterHandler's parameter objects
<code>potential</code>	A descriptor for ParameterType attributes.
<code>version</code>	A descriptor for ParameterType attributes.

```
class AngleType(smirks, allow_cosmetic_attributes=False, **kwargs)
    A SMIRNOFF angle type.
```

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

### Attributes

**angle** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```

>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom

```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
```

(continues on next page)

(continued from previous page)

```
...     # This converter converts only integers to float
...     # and raise an exception for the other types.
...     if isinstance(value, int):
...         return float(value)
...     elif not isinstance(value, float):
...         raise TypeError(f"Cannot convert '{value}' to float")
...     return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**k** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**parent\_id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**smirks** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

**`add_cosmetic_attribute(self, attr_name, attr_value)`**

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**`attr_name`** [str] Name of the attribute to define for this object.

**`attr_value`** [str] The value of the attribute to define for this object.

**`delete_cosmetic_attribute(self, attr_name)`**

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**`attr_name`** [str] Name of the cosmetic attribute to delete.

**`to_dict(self, discard_cosmetic_attributes=False)`**

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

### Parameters

**`discard_cosmetic_attributes`** [bool, optional. Default = `False`] Whether to discard non-spec attributes of this object

**Returns**

**smirnoff\_dict** [dict] The SMIRNOFF-compliant dict representation of this object.

**check\_handler\_compatibility**(*self*, *other\_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

**Parameters**

**other\_handler** [a ParameterHandler object] The handler to compare to.

**Raises**

**IncompatibleParameterError** if **handler\_kwargs** are incompatible with existing parameters.

**add\_cosmetic\_attribute**(*self*, *attr\_name*, *attr\_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

**Parameters**

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

**add\_parameter**(*self*, *parameter\_kwargs*)

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

**Parameters**

**parameter\_kwargs** [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

**assign\_parameters**(*self*, *topology*, *system*)

Assign parameters for the given Topology to the specified System object.

**Parameters**

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

**Parameters**



**attr\_name** [str] Name of the cosmetic attribute to delete.

**find\_matches**(*self*, *entity*)

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

#### Parameters

**entity** [openforcefield.topology.Topology] Topology to search.

#### Returns

**matches** [ValenceDict[Tuple[int], ParameterHandler.Match]]  
 matches[particle\_indices] is the ParameterType object matching the tuple of particle indices in entity.

**get\_parameter**(*self*, *parameter\_attrs*)

Return the parameters in this ParameterHandler that match the parameter\_attrs argument

#### Parameters

**parameter\_attrs** [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[\*:1]~[#16:2]=,.[#6:3]~[:4]", "id": "t105"})

#### Returns

**list of ParameterType objects** A list of matching ParameterType objects

**property known\_kwargs**

List of kwargs that can be parsed by the function.

**property parameters**

The ParameterList that holds this ParameterHandler's parameter objects

**postprocess\_system**(*self*, *topology*, *system*, *\*\*kwargs*)

Allow the force to perform a final post-processing pass on the System following parameter assignment, if needed.

#### Parameters

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**to\_dict**(*self*, *discard\_cosmetic\_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

#### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

#### Returns

**smirnoff\_data** [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

`openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler`

```
class openforcefield.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.

**Attributes**

**default\_idivf** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**known\_kwargs** List of kwargs that can be parsed by the function.

**parameters** The `ParameterList` that holds this `ParameterHandler`'s parameter objects

**potential** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**version** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> 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`

## Methods

<code>ProperTorsionType(smirks[, ...])</code>	A SMIRNOFF torsion type for proper torsions.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the <code>parameter_attrs</code> argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

create\_force

**`__init__(self, allow_cosmetic_attributes=False, skip_version_check=False, **kwargs)`**

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

### Parameters

**`allow_cosmetic_attributes`** [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

**`skip_version_check`**: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to `_MAX_SUPPORTED_SECTION_VERSION`.

**`**kwargs`** [dict] The dict representation of the SMIRNOFF data source

## Methods

<code>__init__(self[, allow_cosmetic_attributes, ...])</code>	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>create_force(self, system, topology, **kwargs)</code>	
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the parameter_attrs argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

## Attributes

<code>default_idivf</code>	A descriptor for ParameterType attributes.
<code>known_kwargs</code>	List of kwargs that can be parsed by the function.
<code>parameters</code>	The ParameterList that holds this ParameterHandler's parameter objects
<code>potential</code>	A descriptor for ParameterType attributes.
<code>version</code>	A descriptor for ParameterType attributes.

**class ProperTorsionType**(*smirks*, *allow\_cosmetic\_attributes=False*, *\*\*kwargs*)  
 A SMIRNOFF torsion type for proper torsions.

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

## Attributes

**id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
```

(continues on next page)

(continued from previous page)

```

...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**idivf** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [`simtk.unit.Quantity`, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```

>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)

```

(continues on next page)

(continued from previous page)

```
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with ParameterAttribute, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**k** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with `ParameterAttribute`, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**parent\_id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**periodicity** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [`simtk.unit.Quantity`, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into `Quantity` objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with `ParameterAttribute`, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**phase** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with ParameterAttribute, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**smirks** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```



`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

**`add_cosmetic_attribute(self, attr_name, attr_value)`**

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**`attr_name`** [str] Name of the attribute to define for this object.

**`attr_value`** [str] The value of the attribute to define for this object.

**`delete_cosmetic_attribute(self, attr_name)`**

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**`attr_name`** [str] Name of the cosmetic attribute to delete.

**`to_dict(self, discard_cosmetic_attributes=False)`**

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

### Parameters

**`discard_cosmetic_attributes`** [bool, optional. Default = `False`] Whether to discard non-spec attributes of this object

**Returns**

**smirnoff\_dict** [dict] The SMIRNOFF-compliant dict representation of this object.

**check\_handler\_compatibility**(*self*, *other\_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

**Parameters**

**other\_handler** [a ParameterHandler object] The handler to compare to.

**Raises**

**IncompatibleParameterError** if **handler\_kwargs** are incompatible with existing parameters.

**add\_cosmetic\_attribute**(*self*, *attr\_name*, *attr\_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

**Parameters**

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

**add\_parameter**(*self*, *parameter\_kwargs*)

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

**Parameters**

**parameter\_kwargs** [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

**assign\_parameters**(*self*, *topology*, *system*)

Assign parameters for the given Topology to the specified System object.

**Parameters**

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

**Parameters**

**attr\_name** [str] Name of the cosmetic attribute to delete.

**find\_matches**(*self*, *entity*)

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

#### Parameters

**entity** [openforcefield.topology.Topology] Topology to search.

#### Returns

**matches** [ValenceDict[Tuple[int], ParameterHandler.Match]]  
 matches[particle\_indices] is the ParameterType object matching the tuple of particle indices in entity.

**get\_parameter**(*self*, *parameter\_attrs*)

Return the parameters in this ParameterHandler that match the parameter\_attrs argument

#### Parameters

**parameter\_attrs** [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[ :1]~[#16:2]=,.[#6:3]~[:4]", "id": "t105"})

#### Returns

**list of ParameterType objects** A list of matching ParameterType objects

**property known\_kwargs**

List of kwargs that can be parsed by the function.

**property parameters**

The ParameterList that holds this ParameterHandler's parameter objects

**postprocess\_system**(*self*, *topology*, *system*, *\*\*kwargs*)

Allow the force to perform a final post-processing pass on the System following parameter assignment, if needed.

#### Parameters

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**to\_dict**(*self*, *discard\_cosmetic\_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

#### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

#### Returns

**smirnoff\_data** [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

`openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler`

```
class openforcefield.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.

**Attributes**

**default\_idivf** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**known\_kwargs** List of kwargs that can be parsed by the function.

**parameters** The `ParameterList` that holds this `ParameterHandler`'s parameter objects

**potential** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**version** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)



(continued from previous page)

```
>>> 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`

## Methods

<code>ImproperTorsionType(smirks[, ...])</code>	A SMIRNOFF torsion type for improper torsions.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the improper torsions in the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the <code>parameter_attrs</code> argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

create\_force

**`__init__(self, allow_cosmetic_attributes=False, skip_version_check=False, **kwargs)`**

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

### Parameters

**allow\_cosmetic\_attributes** [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

**skip\_version\_check**: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to `_MAX_SUPPORTED_SECTION_VERSION`.

**\*\*kwargs** [dict] The dict representation of the SMIRNOFF data source

## Methods

<code>__init__(self[, allow_cosmetic_attributes, ...])</code>	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>create_force(self, system, topology, **kwargs)</code>	
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the improper torsions in the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the parameter_attrs argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

## Attributes

<code>default_idivf</code>	A descriptor for ParameterType attributes.
<code>known_kwargs</code>	List of kwargs that can be parsed by the function.
<code>parameters</code>	The ParameterList that holds this ParameterHandler's parameter objects
<code>potential</code>	A descriptor for ParameterType attributes.
<code>version</code>	A descriptor for ParameterType attributes.

**class ImproperTorsionType**(*smirks*, *allow\_cosmetic\_attributes=False*, *\*\*kwargs*)  
 A SMIRNOFF torsion type for improper torsions.

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

## Attributes

**id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
```

(continues on next page)

(continued from previous page)

```

...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**idivf** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [`simtk.unit.Quantity`, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```

>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)

```

(continues on next page)

(continued from previous page)

```
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with ParameterAttribute, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**k** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [`simtk.unit.Quantity`, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with `ParameterAttribute`, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**parent\_id** A descriptor for `ParameterType` attributes.

The descriptors allow 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**periodicity** 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.

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [`simtk.unit.Quantity`, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into `Quantity` objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with `ParameterAttribute`, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**phase** 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.



**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

**ParameterAttribute** A simple parameter attribute.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with ParameterAttribute, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

**smirks** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

**`add_cosmetic_attribute(self, attr_name, attr_value)`**

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**`attr_name`** [str] Name of the attribute to define for this object.

**`attr_value`** [str] The value of the attribute to define for this object.

**`delete_cosmetic_attribute(self, attr_name)`**

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**`attr_name`** [str] Name of the cosmetic attribute to delete.

**`to_dict(self, discard_cosmetic_attributes=False)`**

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

### Parameters

**`discard_cosmetic_attributes`** [bool, optional. Default = `False`] Whether to discard non-spec attributes of this object

**Returns**

**smirnoff\_dict** [dict] The SMIRNOFF-compliant dict representation of this object.

**check\_handler\_compatibility**(*self*, *other\_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

**Parameters**

**other\_handler** [a ParameterHandler object] The handler to compare to.

**Raises**

**IncompatibleParameterError** if **handler\_kwargs** are incompatible with existing parameters.

**find\_matches**(*self*, *entity*)

Find the improper torsions in the topology/molecule matched by a parameter type.

**Parameters**

**entity** [openforcefield.topology.Topology] Topology to search.

**Returns**

**matches** [ImproperDict[Tuple[int], ParameterHandler.Match]]  
matches[atom\_indices] is the ParameterType object matching the 4-tuple of atom indices in entity.

**add\_cosmetic\_attribute**(*self*, *attr\_name*, *attr\_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

**Parameters**

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

**add\_parameter**(*self*, *parameter\_kwargs*)

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

**Parameters**

**parameter\_kwargs** [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

**assign\_parameters**(*self*, *topology*, *system*)

Assign parameters for the given Topology to the specified System object.

**Parameters**

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)  
Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the cosmetic attribute to delete.

**get\_parameter**(*self*, *parameter\_attrs*)  
Return the parameters in this ParameterHandler that match the *parameter\_attrs* argument

#### Parameters

**parameter\_attrs** [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[ :1]~[#16:2]=,.[#6:3]~[:4]", "id": "t105"} )

#### Returns

**list of ParameterType objects** A list of matching ParameterType objects

**property known\_kwargs**  
List of kwargs that can be parsed by the function.

**property parameters**  
The ParameterList that holds this ParameterHandler's parameter objects

**postprocess\_system**(*self*, *topology*, *system*, *\*\*kwargs*)  
Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.

#### Parameters

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**to\_dict**(*self*, *discard\_cosmetic\_attributes=False*)  
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

#### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

#### Returns

**smirnoff\_data** [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

`openforcefield.typing.engines.smirnoff.parameters.vdWHandler`

```
class openforcefield.typing.engines.smirnoff.parameters.vdWHandler(allow_cosmetic_attributes=False,  
                                                                skip_version_check=False,  
                                                                **kwargs)
```

Handle SMIRNOFF <vdW> tags

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

**Attributes**

**combining\_rules** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:  
...     attr_required = ParameterAttribute()  
...     attr_optional = ParameterAttribute(default=2)  
...  
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional  
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required  
Traceback (most recent call last):  
...  
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**cutoff** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:  
...     attr_required = ParameterAttribute()  
...     attr_optional = ParameterAttribute(default=2)  
...  
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional  
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required  
Traceback (most recent call last):  
...  
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit  
>>> class MyParameter:  
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)  
...  
>>> my_par = MyParameter()  
>>> my_par.attr_quantity = '1.0 * nanometer'  
>>> my_par.attr_quantity  
Quantity(value=1.0, unit=nanometer)  
>>> my_par.attr_quantity = 3.0  
Traceback (most recent call last):  
...  
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_  
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:  
...     # Both strings and integers convert nicely to floats with float().  
...     attr_all_to_float = ParameterAttribute(converter=float)
```

(continues on next page)



(continued from previous page)

```

...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**known\_kwargs** List of kwargs that can be parsed by the function.

**method** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> 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`

**parameters** The `ParameterList` that holds this `ParameterHandler`'s parameter objects

**potential** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [`simtk.unit.Quantity`, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**scale12** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
```

(continues on next page)

(continued from previous page)

```

...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**scale13** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> 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`

**scale14** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.



```

>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom

```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**scale15** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
```

(continues on next page)

(continued from previous page)

```

...     # This converter converts only integers to float
...     # and raise an exception for the other types.
...     if isinstance(value, int):
...         return float(value)
...     elif not isinstance(value, float):
...         raise TypeError(f"Cannot convert '{value}' to float")
...     return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**switch\_width** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```

>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()

```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**version** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3` `>>> my_par.attr_int_to_float 3.0` `>>> my_par.attr_int_to_float = '4.0'` Traceback (most recent call last): ... `TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the <code>parameter_attrs</code> argument

Continued on next page

Table 89 – continued from previous page

<code>postprocess_system(self, system, topology, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.
<code>vdWType(**kwargs)</code>	A SMIRNOFF vdWForce type.

**create\_force**

**`__init__(self, allow_cosmetic_attributes=False, skip_version_check=False, **kwargs)`**  
 Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

### Parameters

**`allow_cosmetic_attributes`** [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

**`skip_version_check`**: bool, optional. **Default = False** If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to `_MAX_SUPPORTED_SECTION_VERSION`.

**`**kwargs`** [dict] The dict representation of the SMIRNOFF data source

### Methods

<code>__init__(self[, allow_cosmetic_attributes, ...])</code>	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>create_force(self, system, topology, \**kwargs)</code>	
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the <code>parameter_attrs</code> argument
<code>postprocess_system(self, system, topology, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

## Attributes

<code>combining_rules</code>	A descriptor for <code>ParameterType</code> attributes.
<code>cutoff</code>	A descriptor for <code>ParameterType</code> attributes.
<code>known_kwargs</code>	List of kwargs that can be parsed by the function.
<code>method</code>	A descriptor for <code>ParameterType</code> attributes.
<code>parameters</code>	The <code>ParameterList</code> that holds this <code>ParameterHandler</code> 's parameter objects
<code>potential</code>	A descriptor for <code>ParameterType</code> attributes.
<code>scale12</code>	A descriptor for <code>ParameterType</code> attributes.
<code>scale13</code>	A descriptor for <code>ParameterType</code> attributes.
<code>scale14</code>	A descriptor for <code>ParameterType</code> attributes.
<code>scale15</code>	A descriptor for <code>ParameterType</code> attributes.
<code>switch_width</code>	A descriptor for <code>ParameterType</code> attributes.
<code>version</code>	A descriptor for <code>ParameterType</code> attributes.

```
class vdWType(**kwargs)
    A SMIRNOFF vdWForce type.
```

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

## Attributes

**epsilon** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.



```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> 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`

**id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```

>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom

```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**parent\_id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↪ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
```

(continues on next page)

(continued from previous page)

```

...     # This converter converts only integers to float
...     # and raise an exception for the other types.
...     if isinstance(value, int):
...         return float(value)
...     elif not isinstance(value, float):
...         raise TypeError(f"Cannot convert '{value}' to float")
...     return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**rmin\_half** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```

>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()

```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**sigma** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**smirks** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.



**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

**`add_cosmetic_attribute(self, attr_name, attr_value)`**

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**`attr_name`** [str] Name of the attribute to define for this object.

**`attr_value`** [str] The value of the attribute to define for this object.

**`delete_cosmetic_attribute(self, attr_name)`**

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

### Parameters

**`attr_name`** [str] Name of the cosmetic attribute to delete.

**`to_dict(self, discard_cosmetic_attributes=False)`**

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

### Parameters

**`discard_cosmetic_attributes`** [bool, optional. Default = `False`] Whether to discard non-spec attributes of this object

**Returns**

**smirnoff\_dict** [dict] The SMIRNOFF-compliant dict representation of this object.

**check\_handler\_compatibility**(*self*, *other\_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

**Parameters**

**other\_handler** [a ParameterHandler object] The handler to compare to.

**Raises**

**IncompatibleParameterError** if **handler\_kwargs** are incompatible with existing parameters.

**postprocess\_system**(*self*, *system*, *topology*, *\*\*kwargs*)

Allow the force to perform a final post-processing pass on the System following parameter assignment, if needed.

**Parameters**

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**add\_cosmetic\_attribute**(*self*, *attr\_name*, *attr\_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

**Parameters**

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

**add\_parameter**(*self*, *parameter\_kwargs*)

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

**Parameters**

**parameter\_kwargs** [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

**assign\_parameters**(*self*, *topology*, *system*)

Assign parameters for the given Topology to the specified System object.

**Parameters**

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)  
Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the cosmetic attribute to delete.

**find\_matches**(*self*, *entity*)  
Find the elements of the topology/molecule matched by a parameter type.

#### Parameters

**entity** [openforcefield.topology.Topology] Topology to search.

#### Returns

**matches** [ValenceDict[Tuple[int], ParameterHandler.\_Match]]  
matches[particle\_indices] is the ParameterType object matching the tuple of particle indices in entity.

**get\_parameter**(*self*, *parameter\_attrs*)  
Return the parameters in this ParameterHandler that match the parameter\_attrs argument

#### Parameters

**parameter\_attrs** [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[ :1]~[#16:2]=,:[#6:3]~[:4]", "id": "t105"} )

#### Returns

**list of ParameterType objects** A list of matching ParameterType objects

**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**(*self*, *discard\_cosmetic\_attributes=False*)  
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

#### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

#### Returns

**smirnoff\_data** [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

`openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler`

```
class openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler(allow_cosmetic_attributes=False,
                                                                              skip_version_check=False,
                                                                              **kwargs)
```

Handles SMIRNOFF <Electrostatics> tags.

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

**Attributes**

**cutoff** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**known\_kwargs** List of kwargs that can be parsed by the function.

**method** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
```

(continues on next page)

(continued from previous page)

```

...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**parameters** The `ParameterList` that holds this `ParameterHandler`'s parameter objects

**scale12** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.



```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> 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`

**scale13** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```

>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom

```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**scale14** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
```

(continues on next page)

(continued from previous page)

```

...     # This converter converts only integers to float
...     # and raise an exception for the other types.
...     if isinstance(value, int):
...         return float(value)
...     elif not isinstance(value, float):
...         raise TypeError(f"Cannot convert '{value}' to float")
...     return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**scale15** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```

>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()

```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**switch\_width** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**version** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.



**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the <code>parameter_attrs</code> argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

**create\_force**

`__init__(self, allow_cosmetic_attributes=False, skip_version_check=False, **kwargs)`  
Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

## Parameters

**allow\_cosmetic\_attributes** [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

**skip\_version\_check**: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to `_MAX_SUPPORTED_SECTION_VERSION`.

**\*\*kwargs** [dict] The dict representation of the SMIRNOFF data source

## Methods

<code>__init__(self[, allow_cosmetic_attributes, ...])</code>	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>create_force(self, system, topology,   \**kwargs)</code>	
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the parameter_attrs argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

## Attributes

<code>cutoff</code>	A descriptor for ParameterType attributes.
<code>known_kwargs</code>	List of kwargs that can be parsed by the function.
<code>method</code>	A descriptor for ParameterType attributes.
<code>parameters</code>	The ParameterList that holds this ParameterHandler's parameter objects
<code>scale12</code>	A descriptor for ParameterType attributes.
<code>scale13</code>	A descriptor for ParameterType attributes.
<code>scale14</code>	A descriptor for ParameterType attributes.
<code>scale15</code>	A descriptor for ParameterType attributes.
<code>switch_width</code>	A descriptor for ParameterType attributes.
<code>version</code>	A descriptor for ParameterType attributes.

**`check_handler_compatibility(self, other_handler)`**

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

### Parameters

**`other_handler`** [a ParameterHandler object] The handler to compare to.

### Raises

**IncompatibleParameterError** if handler\_kwargs are incompatible with existing parameters.

**`add_cosmetic_attribute(self, attr_name, attr_value)`**

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

**add\_parameter**(*self*, *parameter\_kwargs*)

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

#### Parameters

**parameter\_kwargs** [dict] The kwargs to pass to the `ParameterHandler.INFO_TYPE` (a `ParameterType`) constructor

**assign\_parameters**(*self*, *topology*, *system*)

Assign parameters for the given Topology to the specified System object.

#### Parameters

**topology** [`openforcefield.topology.Topology`] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [`simtk.openmm.System`] The OpenMM System object to add the Force (or append new parameters) to.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the cosmetic attribute to delete.

**find\_matches**(*self*, *entity*)

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

#### Parameters

**entity** [`openforcefield.topology.Topology`] Topology to search.

#### Returns

**matches** [`ValenceDict[Tuple[int], ParameterHandler._Match]`]  
matches[particle\_indices] is the `ParameterType` object matching the tuple of particle indices in entity.

**get\_parameter**(*self*, *parameter\_attrs*)

Return the parameters in this `ParameterHandler` that match the `parameter_attrs` argument

#### Parameters

**parameter\_attrs** [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[\*:1]~[#16:2]=,.[#6:3]~[:4]", "id": "t105"})

#### Returns

**list of ParameterType objects** A list of matching ParameterType objects

#### property known\_kwargs

List of kwargs that can be parsed by the function.

#### property parameters

The ParameterList that holds this ParameterHandler's parameter objects

#### postprocess\_system(self, topology, system, \*\*kwargs)

Allow the force to perform a final post-processing pass on the System following parameter assignment, if needed.

#### Parameters

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

#### to\_dict(self, discard\_cosmetic\_attributes=False)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

#### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

#### Returns

**smirnoff\_data** [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

### openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler

```
class openforcefield.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.

#### Attributes

**known\_kwargs** List of kwargs that can be parsed by the function.

**parameters** The ParameterList that holds this ParameterHandler's parameter objects

**version** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0'` Traceback (most recent call last): ... `TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_charge_from_molecules(self, molecule, ...)</code>	Given an input molecule, checks against a list of molecules for an isomorphic match.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the <code>parameter_attrs</code> argument
<code>postprocess_system(self, system, topology, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

create_force	
--------------	--

**\_\_init\_\_**(self, allow\_cosmetic\_attributes=False, skip\_version\_check=False, \*\*kwargs)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

### Parameters

**allow\_cosmetic\_attributes** [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

**skip\_version\_check**: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to `_MAX_SUPPORTED_SECTION_VERSION`.

**\*\*kwargs** [dict] The dict representation of the SMIRNOFF data source

### Methods

<code>__init__(self[, allow_cosmetic_attributes, ...])</code>	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_charge_from_molecules(self, molecule, ...)</code>	Given an input molecule, checks against a list of molecules for an isomorphic match.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>create_force(self, system, topology, \*\*kwargs)</code>	
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the parameter_attrs argument
<code>postprocess_system(self, system, topology, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

### Attributes

<code>known_kwargs</code>	List of kwargs that can be parsed by the function.
<code>parameters</code>	The ParameterList that holds this ParameterHandler's parameter objects
<code>version</code>	A descriptor for ParameterType attributes.



**check\_handler\_compatibility**(*self*, *other\_handler*, *assume\_missing\_is\_default=True*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

#### Parameters

**other\_handler** [a ParameterHandler object] The handler to compare to.

#### Raises

**IncompatibleParameterError** if **handler\_kwargs** are incompatible with existing parameters.

**assign\_charge\_from\_molecules**(*self*, *molecule*, *charge\_mols*)

Given an input molecule, checks against a list of molecules for an isomorphic match. If found, assigns partial charges from the match to the input molecule.

#### Parameters

**molecule** [an openforcefield.topology.FrozenMolecule] The molecule to have partial charges assigned if a match is found.

**charge\_mols** [list of [openforcefield.topology.FrozenMolecule]] A list of molecules with charges already assigned.

#### Returns

**match\_found** [bool] Whether a match was found. If True, the input molecule will have been modified in-place.

**postprocess\_system**(*self*, *system*, *topology*, *\*\*kwargs*)

Allow the force to perform a final post-processing pass on the System following parameter assignment, if needed.

#### Parameters

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**add\_cosmetic\_attribute**(*self*, *attr\_name*, *attr\_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the attribute to define for this object.

**attr\_value** [str] The value of the attribute to define for this object.

**add\_parameter**(*self*, *parameter\_kwargs*)

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

#### Parameters

**parameter\_kwargs** [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

**assign\_parameters**(*self*, *topology*, *system*)

Assign parameters for the given Topology to the specified System object.

#### Parameters

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)

Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the cosmetic attribute to delete.

**find\_matches**(*self*, *entity*)

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

#### Parameters

**entity** [openforcefield.topology.Topology] Topology to search.

#### Returns

**matches** [ValenceDict[Tuple[int], ParameterHandler.\_Match]] matches[particle\_indices] is the ParameterType object matching the tuple of particle indices in entity.

**get\_parameter**(*self*, *parameter\_attrs*)

Return the parameters in this ParameterHandler that match the parameter\_attrs argument

#### Parameters

**parameter\_attrs** [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[ :1]~[#16:2]=,:[#6:3]~[:4]", "id": "t105"} )

#### Returns

**list of ParameterType objects** A list of matching ParameterType objects

**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**(*self*, *discard\_cosmetic\_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

#### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

#### Returns

**smirnoff\_data** [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

### openforcefield.typing.engines.smirnoff.parameters.GBSAHandler

```
class openforcefield.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.

#### Attributes

**gb\_model** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**known\_kwargs** List of kwargs that can be parsed by the function.

**parameters** The ParameterList that holds this ParameterHandler's parameter objects

**sa\_model** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
```

(continues on next page)

(continued from previous page)

```
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**solute\_dielectric** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
```

(continues on next page)

(continued from previous page)

```

...         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`

**`solvent_dielectric`** 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).

**`default`** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**`unit`** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**`converter`** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**`IndexedParameterAttribute`** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```

>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()

```

Even without explicit assignment, the default value is returned.

```

>>> my_par.attr_optional
2

```



If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**solvent\_radius** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**surface\_area\_penalty** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> 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`

**version** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```

>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom

```

You can attach a custom converter to an attribute.

```

>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

## Methods

<code>GBSAType(**kwargs)</code>	A SMIRNOFF GBSA type.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.

Continued on next page

Table 99 – continued from previous page

<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the <code>parameter_attrs</code> argument
<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

create\_force

`__init__(self, allow_cosmetic_attributes=False, skip_version_check=False, **kwargs)`  
 Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

#### Parameters

**allow\_cosmetic\_attributes** [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

**skip\_version\_check**: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to `_MAX_SUPPORTED_SECTION_VERSION`.

**\*\*kwargs** [dict] The dict representation of the SMIRNOFF data source

#### Methods

<code>__init__(self[, allow_cosmetic_attributes, ...])</code>	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>add_parameter(self, parameter_kwargs)</code>	Add a parameter to the forcefield, ensuring all parameters are valid.
<code>assign_parameters(self, topology, system)</code>	Assign parameters for the given Topology to the specified System object.
<code>check_handler_compatibility(self, other_handler)</code>	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
<code>create_force(self, system, topology, \**kwargs)</code>	
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>find_matches(self, entity)</code>	Find the elements of the topology/molecule matched by a parameter type.
<code>get_parameter(self, parameter_attrs)</code>	Return the parameters in this ParameterHandler that match the <code>parameter_attrs</code> argument

Continued on next page

Table 100 – continued from previous page

<code>postprocess_system(self, topology, system, ...)</code>	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this ParameterHandler to an Ordered-Dict, compliant with the SMIRNOFF data spec.

### Attributes

<code>gb_model</code>	A descriptor for ParameterType attributes.
<code>known_kwargs</code>	List of kwargs that can be parsed by the function.
<code>parameters</code>	The ParameterList that holds this ParameterHandler's parameter objects
<code>sa_model</code>	A descriptor for ParameterType attributes.
<code>solute_dielectric</code>	A descriptor for ParameterType attributes.
<code>solvent_dielectric</code>	A descriptor for ParameterType attributes.
<code>solvent_radius</code>	A descriptor for ParameterType attributes.
<code>surface_area_penalty</code>	A descriptor for ParameterType attributes.
<code>version</code>	A descriptor for ParameterType attributes.

```
class GBSAType(**kwargs)
    A SMIRNOFF GBSA type.
```

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

### Attributes

**id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.



Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
```

(continues on next page)

(continued from previous page)

```

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`

**parent\_id** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [`simtk.unit.Quantity`, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```

>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()

```

Even without explicit assignment, the default value is returned.

```

>>> my_par.attr_optional
2

```

If you try to access an attribute without setting it first, an exception is raised.

```

>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'

```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**radius** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
```

(continues on next page)

(continued from previous page)

```

...     # This converter converts only integers to float
...     # and raise an exception for the other types.
...     if isinstance(value, int):
...         return float(value)
...     elif not isinstance(value, float):
...         raise TypeError(f"Cannot convert '{value}' to float")
...     return value
...
>>> my_par = MyParameter()

```

`attr_all_to_float` accepts and convert to float both strings and integers

```

>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0

```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**scale** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```

>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()

```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

**smirks** 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).

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**IndexedParameterAttribute** A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0_
↳ dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

`attr_all_to_float` accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to `attr_int_to_float` converts only integers instead. `>>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float`

## Methods

<code>add_cosmetic_attribute(self, attr_name, ...)</code>	Add a cosmetic attribute to this object.
<code>delete_cosmetic_attribute(self, attr_name)</code>	Delete a cosmetic attribute from this object.
<code>to_dict(self[, discard_cosmetic_attributes])</code>	Convert this object to dict format.

**add\_cosmetic\_attribute**(*self*, *attr\_name*, *attr\_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).



**Parameters**

**attr\_name** [str] Name of the attribute to define for this object.  
**attr\_value** [str] The value of the attribute to define for this object.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)  
 Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

**Parameters**

**attr\_name** [str] Name of the cosmetic attribute to delete.

**to\_dict**(*self*, *discard\_cosmetic\_attributes=False*)  
 Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if *discard\_cosmetic\_attributes* is False.

**Parameters**

**discard\_cosmetic\_attributes** [bool, optional. Default = False] Whether to discard non-spec attributes of this object

**Returns**

**smirnoff\_dict** [dict] The SMIRNOFF-compliant dict representation of this object.

**add\_cosmetic\_attribute**(*self*, *attr\_name*, *attr\_value*)  
 Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

**Parameters**

**attr\_name** [str] Name of the attribute to define for this object.  
**attr\_value** [str] The value of the attribute to define for this object.

**add\_parameter**(*self*, *parameter\_kwargs*)  
 Add a parameter to the forcefield, ensuring all parameters are valid.

**Parameters**

**parameter\_kwargs** [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

**assign\_parameters**(*self*, *topology*, *system*)  
 Assign parameters for the given Topology to the specified System object.

**Parameters**

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**delete\_cosmetic\_attribute**(*self*, *attr\_name*)  
Delete a cosmetic attribute from this object.

**Warning:** The API for modifying cosmetic attributes is experimental

and may change in the future (see issue #338).

#### Parameters

**attr\_name** [str] Name of the cosmetic attribute to delete.

**find\_matches**(*self*, *entity*)  
Find the elements of the topology/molecule matched by a parameter type.

#### Parameters

**entity** [openforcefield.topology.Topology] Topology to search.

#### Returns

**matches** [ValenceDict[Tuple[int], ParameterHandler.\_Match]]  
matches[particle\_indices] is the ParameterType object matching the tuple of particle indices in entity.

**get\_parameter**(*self*, *parameter\_attrs*)  
Return the parameters in this ParameterHandler that match the parameter\_attrs argument

#### Parameters

**parameter\_attrs** [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[ :1]~[#16:2]=,:[#6:3]~[:4]", "id": "t105"} )

#### Returns

**list of ParameterType objects** A list of matching ParameterType objects

**property known\_kwargs**  
List of kwargs that can be parsed by the function.

**property parameters**  
The ParameterList that holds this ParameterHandler's parameter objects

**postprocess\_system**(*self*, *topology*, *system*, *\*\*kwargs*)  
Allow the force to perform a final post-processing pass on the System following parameter assignment, if needed.

#### Parameters

**topology** [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

**system** [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

**to\_dict**(*self*, *discard\_cosmetic\_attributes=False*)  
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

#### Parameters

**discard\_cosmetic\_attributes** [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

**Returns**

**smirnoff\_data** [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

**check\_handler\_compatibility**(self, other\_handler)  
Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

**Parameters**

**other\_handler** [a ParameterHandler object] The handler to compare to.

**Raises**

**IncompatibleParameterError** if handler\_kwargs are incompatible with existing parameters.

Parameter I/O Handlers

ParameterIOHandler objects handle reading and writing of serialized 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.

openforcefield.typing.engines.smirnoff.io.ParameterIOHandler

**class** openforcefield.typing.engines.smirnoff.io.**ParameterIOHandler**  
Base class for handling serialization/deserialization of SMIRNOFF ForceField objects

**Methods**

parse_file(self, file_path)	
	<b>Parameters</b>
parse_string(self, data)	Parse a SMIRNOFF force field definition in a serialized format
to_file(self, file_path, smirnoff_data)	Write the current forcefield parameter set to a file.
to_string(self, smirnoff_data)	Render the forcefield parameter set to a string

**\_\_init\_\_**(self)  
Create a new ParameterIOHandler.

**Methods**

<code>__init__(self)</code>	Create a new ParameterIOHandler.
<code>parse_file(self, file_path)</code>	
<b>Parameters</b>	
<code>parse_string(self, data)</code>	Parse a SMIRNOFF force field definition in a serialized format
<code>to_file(self, file_path, smirnoff_data)</code>	Write the current forcefield parameter set to a file.
<code>to_string(self, smirnoff_data)</code>	Render the forcefield parameter set to a string

`parse_file(self, file_path)`

#### Parameters

**file\_path**

`parse_string(self, data)`

Parse a SMIRNOFF force field definition in a serialized format

#### Parameters

**data**

`to_file(self, file_path, smirnoff_data)`

Write the current forcefield parameter set to a file.

#### Parameters

**file\_path** [str] The path to the file to write to.

**smirnoff\_data** [dict] A dictionary structured in compliance with the SMIRNOFF spec

`to_string(self, smirnoff_data)`

Render the forcefield parameter set to a string

#### Parameters

**smirnoff\_data** [dict] A dictionary structured in compliance with the SMIRNOFF spec

#### Returns

**str**

## openforcefield.typing.engines.smirnoff.io.XMLParameterIOHandler

**class** openforcefield.typing.engines.smirnoff.io.XMLParameterIOHandler

Handles serialization/deserialization of SMIRNOFF ForceField objects from OFFXML format.

#### Methods

<code>parse_file(self, source)</code>	Parse a SMIRNOFF force field definition in XML format, read from a file.
<code>parse_string(self, data)</code>	Parse a SMIRNOFF force field definition in XML format.
<code>to_file(self, file_path, smirnoff_data)</code>	Write the current forcefield parameter set to a file.

Continued on next page

Table 106 – continued from previous page

<code>to_string(self, smirnoff_data)</code>	Write the current forcefield parameter set to an XML string.
---	--

`__init__(self)`  
Create a new ParameterIOHandler.

## Methods

<code>__init__(self)</code>	Create a new ParameterIOHandler.
<code>parse_file(self, source)</code>	Parse a SMIRNOFF force field definition in XML format, read from a file.
<code>parse_string(self, data)</code>	Parse a SMIRNOFF force field definition in XML format.
<code>to_file(self, file_path, smirnoff_data)</code>	Write the current forcefield parameter set to a file.
<code>to_string(self, smirnoff_data)</code>	Write the current forcefield parameter set to an XML string.

`parse_file(self, source)`  
Parse a SMIRNOFF force field definition in XML format, read from a file.

### Parameters

**source** [str or io.RawIOBase] File path of file-like object implementing a `read()` method specifying a SMIRNOFF force field definition in [the SMIRNOFF XML format](#).

### Raises

**ParseError** If the XML cannot be processed.

**FileNotFoundError** If the file could not found.

`parse_string(self, data)`  
Parse a SMIRNOFF force field definition in XML format.  
A `ParseError` is raised if the XML cannot be processed.

### Parameters

**data** [str] A SMIRNOFF force field definition in [the SMIRNOFF XML format](#).

`to_file(self, file_path, smirnoff_data)`  
Write the current forcefield parameter set to a file.

### Parameters

**file\_path** [str] The path to the file to be written. The `.offxml` or `.xml` file extension must be present.

**smirnoff\_data** [dict] A dict structured in compliance with the SMIRNOFF data spec.

`to_string(self, smirnoff_data)`  
Write the current forcefield parameter set to an XML string.

### Parameters

**smirnoff\_data** [dict] A dictionary structured in compliance with the SMIRNOFF spec

### Returns

**serialized\_forcefield** [str] XML String representation of this forcefield.

### 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.

### openforcefield.typing.engines.smirnoff.parameters.ParameterAttribute

```
class openforcefield.typing.engines.smirnoff.parameters.ParameterAttribute(default=<class  
                                'openforce-  
                                field.typing.engines.smirnoff.parameters  
                                unit=None,  
                                con-  
                                verter=None)
```

A descriptor for ParameterType attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have None as a default value, required attributes have the default set to the special type UNDEFINED.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted\_value converter(instance, parameter\_attribute, value): -> converted\_value

A decorator syntax is available (see example below).

#### Parameters

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

**converter** [callable, optional] An optional function that can be used to convert values before setting the attribute.

**See also:**

**IndexedParameterAttribute** A parameter attribute with multiple terms.

### Examples

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0 dimensionless should have_
↳ units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr\_all\_to\_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> 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`

### Attributes

**name**

### Methods

<code>UNDEFINED</code>	Custom type used by <code>ParameterAttribute</code> to differentiate between <code>None</code> and undeclared default.
<code>converter(self, converter)</code>	Create a new <code>ParameterAttribute</code> with an associated converter.

`__init__(self, default=<class 'openforcefield.typing.engines.smirnoff.parameters.ParameterAttribute.UNDEFINED'>, unit=None, converter=None)`  
 Initialize self. See `help(type(self))` for accurate signature.

### Methods

<code>__init__(self[, default, unit, converter])</code>	Initialize self.
<code>converter(self, converter)</code>	Create a new <code>ParameterAttribute</code> with an associated converter.

### Attributes

<b>name</b>
-------------

### **class UNDEFINED**

Custom type used by `ParameterAttribute` to differentiate between `None` and undeclared default.

### **converter(self, converter)**

Create a new `ParameterAttribute` with an associated converter.

This is meant to be used as a decorator (see main examples).



**openforcefield.typing.engines.smirnoff.parameters.IndexedParameterAttribute**

```
class openforcefield.typing.engines.smirnoff.parameters.IndexedParameterAttribute(default=<class
                                                    'open-
                                                    force-
                                                    field.typing.engines.smirnoff.pa
                                                    unit=None,
                                                    con-
                                                    verter=None)
```

The attribute of a parameter with an unspecified number of terms.

Some parameters can be associated to multiple terms, For example, torsions have parameters such as `k1`, `k2`, ..., and `IndexedParameterAttribute` can be used to encapsulate the sequence of terms.

The only substantial difference with `ParameterAttribute` is that only sequences are supported as values and converters and units are checked on each element of the sequence.

Currently, the descriptor makes the sequence immutable. This is to avoid that an element of the sequence could be set without being properly validated. In the future, the data could be wrapped in a safe list that would safely allow mutability.

**Parameters**

**default** [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

**unit** [simtk.unit.Quantity, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

**converter** [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

See also:

**ParameterAttribute** A simple parameter attribute.

**Examples**

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with `ParameterAttribute`, but they are used to validate each value in the sequence.

```

>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]

```

## Attributes

**name**

## Methods

<code>UNDEFINED</code>	Custom type used by ParameterAttribute to differentiate between None and undeclared default.
<code>converter(self, converter)</code>	Create a new ParameterAttribute with an associated converter.

`__init__(self, default=<class 'openforcefield.typing.engines.smirnoff.parameters.ParameterAttribute.UNDEFINED'>, unit=None, converter=None)`  
 Initialize self. See help(type(self)) for accurate signature.

## Methods

<code>__init__(self[, default, unit, converter])</code>	Initialize self.
<code>converter(self, converter)</code>	Create a new ParameterAttribute with an associated converter.

## Attributes

<b>name</b>
-------------

**class UNDEFINED**  
 Custom type used by ParameterAttribute to differentiate between None and undeclared default.

**converter(self, converter)**  
 Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

## 2.3 Utilities

### 2.3.1 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 openforcefield.utils.toolkits import ToolkitRegistry
toolkit_registry = ToolkitRegistry()
toolkit_precedence = [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper]
[ toolkit_registry.register(toolkit) for toolkit in toolkit_precedence if toolkit.is_available() ]
```

Alternatively, the global toolkit registry (which will attempt to register any available toolkits) can be used:

```
from openforcefield.utils.toolkits import DEFAULT_TOOLKIT_REGISTRY as toolkit_registry
```

The toolkit wrappers can then be accessed through the registry:

```
molecule = Molecule.from_smiles('Cc1ccccc1')
smiles = toolkit_registry.call('to_smiles', molecule)
```

ToolkitRegistry	Registry for ToolkitWrapper objects
ToolkitWrapper	Toolkit wrapper base class.
OpenEyeToolkitWrapper	OpenEye toolkit wrapper
RDKitToolkitWrapper	RDKit toolkit wrapper
AmberToolsToolkitWrapper	AmberTools toolkit wrapper

### openforcefield.utils.toolkits.ToolkitRegistry

```
class openforcefield.utils.toolkits.ToolkitRegistry(register_imported_toolkit_wrappers=False,
                                                    toolkit_precedence=None,          excep-
                                                    tion_if_unavailable=True)
    Registry for ToolkitWrapper objects
```

#### Examples

Register toolkits in a specified order, skipping if unavailable

```
>>> from openforcefield.utils.toolkits import ToolkitRegistry
>>> toolkit_registry = ToolkitRegistry()
>>> toolkit_precedence = [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper]
>>> for toolkit in toolkit_precedence:
...     if toolkit.is_available():
...         toolkit_registry.register_toolkit(toolkit)
```

Register specified toolkits, raising an exception if one is unavailable

```
>>> toolkit_registry = ToolkitRegistry()
>>> toolkits = [OpenEyeToolkitWrapper, AmberToolsToolkitWrapper]
>>> for toolkit in toolkits:
...     toolkit_registry.register_toolkit(toolkit)
```

Register all available toolkits in arbitrary order

```
>>> from openforcefield.utils import all_subclasses
>>> toolkits = all_subclasses(ToolkitWrapper)
>>> for toolkit in toolkit_precedence:
```

(continues on next page)

(continued from previous page)

```
...     if toolkit.is_available():
...         toolkit_registry.register_toolkit(toolkit)
```

Retrieve the global singleton toolkit registry, which is created when this module is imported from all available toolkits:

```
>>> from openforcefield.utils.toolkits import GLOBAL_TOOLKIT_REGISTRY as toolkit_registry
>>> available_toolkits = toolkit_registry.registered_toolkits
```

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

### Attributes

**registered\_toolkits** List registered toolkits.

### Methods

<code>add_toolkit(self, toolkit_wrapper)</code>	Append a ToolkitWrapper onto the list of toolkits in this ToolkitRegistry
<code>call(self, method_name, *args, **kwargs)</code>	Execute the requested method by attempting to use all registered toolkits in order of precedence.
<code>register_toolkit(self, toolkit_wrapper[, ...])</code>	Register the provided toolkit wrapper class, instantiating an object of it.
<code>resolve(self, method_name)</code>	Resolve the requested method name by checking all registered toolkits in order of precedence for one that provides the requested method.

`__init__(self, register_imported_toolkit_wrappers=False, toolkit_precedence=None, exception_if_unavailable=True)`  
Create an empty toolkit registry.

### Parameters

**register\_imported\_toolkit\_wrappers** [bool, optional, default=False]

If True, will attempt to register all imported ToolkitWrapper subclasses that can be found, in no particular order.

**toolkit\_precedence** [list, optional, default=None] List of toolkit wrapper classes, in order of desired precedence when performing molecule operations. If None, defaults to [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper].

**exception\_if\_unavailable** [bool, optional, default=True] If True, an exception will be raised if the toolkit is unavailable

### Methods

<code>__init__(self[, ...])</code>	Create an empty toolkit registry.
------------------------------------	-----------------------------------

Continued on next page

Table 117 – continued from previous page

<code>add_toolkit(self, toolkit_wrapper)</code>	Append a ToolkitWrapper onto the list of toolkits in this ToolkitRegistry
<code>call(self, method_name, <i>\*</i>args, <i>\*</i>kwargs)</code>	Execute the requested method by attempting to use all registered toolkits in order of precedence.
<code>register_toolkit(self, toolkit_wrapper[, ...])</code>	Register the provided toolkit wrapper class, instantiating an object of it.
<code>resolve(self, method_name)</code>	Resolve the requested method name by checking all registered toolkits in order of precedence for one that provides the requested method.

### Attributes

<code>registered_toolkits</code>	List registered toolkits.
----------------------------------	---------------------------

#### property `registered_toolkits`

List registered toolkits.

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

### Returns

**toolkits** [iterable of toolkit objects]

#### `register_toolkit(self, toolkit_wrapper, exception_if_unavailable=True)`

Register the provided toolkit wrapper class, instantiating an object of it.

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

### Parameters

**toolkit\_wrapper** [instance or subclass of ToolkitWrapper] The toolkit wrapper to register or its class.

**exception\_if\_unavailable** [bool, optional, default=True] If True, an exception will be raised if the toolkit is unavailable

#### `add_toolkit(self, toolkit_wrapper)`

Append a ToolkitWrapper onto the list of toolkits in this ToolkitRegistry

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

### Parameters

**toolkit\_wrapper** [openforcefield.utils.ToolkitWrapper] The ToolkitWrapper object to add to the list of registered toolkits

#### `resolve(self, method_name)`

Resolve the requested method name by checking all registered toolkits in order of precedence for one that provides the requested method.

**Parameters**

**method\_name** [str] The name of the method to resolve

**Returns**

**method** The method of the first registered toolkit that provides the requested method name

**Raises**

**NotImplementedError** if the requested method cannot be found among the registered toolkits

**Examples**

Create a molecule, and call the toolkit to\_smiles() method directly

```
>>> from openforcefield.topology import Molecule
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
>>> toolkit_registry = ToolkitRegistry(register_imported_toolkit_wrappers=True)
>>> method = toolkit_registry.resolve('to_smiles')
>>> smiles = method(molecule)
```

**call**(self, method\_name, \*args, \*\*kwargs)

Execute the requested method by attempting to use all registered toolkits in order of precedence.

\*args and \*\*kwargs are passed to the desired method, and return values of the method are returned

This is a convenient shorthand for toolkit\_registry.resolve\_method(method\_name)(\*args, \*\*kwargs)

**Parameters**

**method\_name** [str] The name of the method to execute

**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 openforcefield.topology import Molecule
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
>>> toolkit_registry = ToolkitRegistry(register_imported_toolkit_wrappers=True)
>>> smiles = toolkit_registry.call('to_smiles', molecule)
```

**openforcefield.utils.toolkits.ToolkitWrapper**

**class** openforcefield.utils.toolkits.**ToolkitWrapper**

Toolkit wrapper base class.

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

### Attributes

**toolkit\_file\_read\_formats** List of file formats that this toolkit can read.

**toolkit\_file\_write\_formats** List of file formats that this toolkit can write.

**toolkit\_installation\_instructions** classmethod(function) -> method

**toolkit\_name** The name of the toolkit wrapped by this class.

### Methods

<code>from_file(self, file_path, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file using this toolkit.
<code>from_file_obj(self, file_obj, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file-like object (an object with a “ <code>.read()</code> ” method using this
<code>is_available()</code>	Check whether the corresponding toolkit can be imported
<code>requires_toolkit()</code>	

**`__init__(self, /, *args, **kwargs)`**  
 Initialize self. See `help(type(self))` for accurate signature.

### Methods

<b><code>__init__(self, /, *args, **kwargs)</code></b>	Initialize self.
<code>from_file(self, file_path, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file using this toolkit.
<code>from_file_obj(self, file_obj, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file-like object (an object with a “ <code>.read()</code> ” method using this
<code>is_available()</code>	Check whether the corresponding toolkit can be imported
<code>requires_toolkit()</code>	

### Attributes

<code>toolkit_file_read_formats</code>	List of file formats that this toolkit can read.
<code>toolkit_file_write_formats</code>	List of file formats that this toolkit can write.
<code>toolkit_installation_instructions</code>	classmethod(function) -> method
<code>toolkit_name</code>	The name of the toolkit wrapped by this class.

**property `toolkit_name`**  
 The name of the toolkit wrapped by this class.

**property `toolkit_installation_instructions`**  
 classmethod(function) -> method

Convert a function to be a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C: @classmethod def f(cls, arg1, arg2, ...):  
    ...
```

It can be called either on the class (e.g. `C.f()`) or on an instance (e.g. `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see the `staticmethod` builtin.

**property `toolkit_file_read_formats`**

List of file formats that this toolkit can read.

**property `toolkit_file_write_formats`**

List of file formats that this toolkit can write.

**static `is_available()`**

Check whether the corresponding toolkit can be imported

**Returns**

**`is_installed`** [bool] True if corresponding toolkit is installed, False otherwise.

**`from_file(self, file_path, file_format, allow_undefined_stereo=False)`**

Return an `openforcefield.topology.Molecule` from a file using this toolkit.

**Parameters**

**`file_path`** [str] The file to read the molecule from

**`file_format`** [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check `ToolkitWrapper.toolkit_file_read_formats` for details.

**`allow_undefined_stereo`** [bool, default=False] If false, raises an exception if any molecules contain undefined stereochemistry.

**Returns**

\_\_\_\_\_

**`molecules`** [Molecule or list of Molecules] a list of Molecule objects is returned.

**`from_file_obj(self, file_obj, file_format, allow_undefined_stereo=False)`**

Return an `openforcefield.topology.Molecule` from a file-like object (an object with a `“read()”` method using toolkit.

**Parameters**

**`file_obj`** [file-like object] The file-like object to read the molecule from

**`file_format`** [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check `ToolkitWrapper.toolkit_file_read_formats` for details.

**`allow_undefined_stereo`** [bool, default=False] If false, raises an exception if any molecules contain undefined stereochemistry. If false, the function skips loading the molecule.



### Returns

**molecules** [Molecule or list of Molecules] a list of Molecule objects is returned.

### openforcefield.utils.toolkits.OpenEyeToolkitWrapper

**class** openforcefield.utils.toolkits.**OpenEyeToolkitWrapper**  
OpenEye toolkit wrapper

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

### Attributes

**toolkit\_file\_read\_formats** List of file formats that this toolkit can read.  
**toolkit\_file\_write\_formats** List of file formats that this toolkit can write.  
**toolkit\_installation\_instructions** classmethod(function) -> method  
**toolkit\_name** The name of the toolkit wrapped by this class.

### Methods

<code>compute_partial_charges(self, molecule[, ...])</code>	Compute partial charges with OpenEye quacpac
<code>compute_partial_charges_am1bcc(self, molecule)</code>	Compute AM1BCC partial charges with OpenEye quacpac.
<code>compute_wiberg_bond_orders(self, molecule[, ...])</code>	Update and store list of bond orders this molecule.
<code>find_smarts_matches(self, molecule, smarts)</code>	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
<code>from_file(self, file_path, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file using this toolkit.
<code>from_file_obj(self, file_obj, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file-like object (an object with a “.read()” method using this toolkit.
<code>from_object(self, object[, ...])</code>	If given an OEMol (or OEMol-derived object), this function will load it into an openforcefield.topology.molecule Otherwise, it will return False.
<code>from_openeye(oemol[, low_undefined_stereo])</code>	al- Create a Molecule from an OpenEye molecule.
<code>from_smiles(self, smiles[, ...])</code>	Create a Molecule from a SMILES string using the OpenEye toolkit.
<code>generate_conformers(self, molecule[, ...])</code>	Generate molecule conformers using OpenEye Omega.
<code>is_available([oetools])</code>	Check if the given OpenEye toolkit components are available.
<code>requires_toolkit()</code>	
<code>to_file(self, molecule, file_path, file_format)</code>	Writes an OpenFF Molecule to a file-like object
<code>to_file_obj(self, molecule, file_obj, ...)</code>	Writes an OpenFF Molecule to a file-like object

Continued on next page

Table 122 – continued from previous page

<code>to_openeye(molecule[, aromaticity_model])</code>	Create an OpenEye molecule using the specified aromaticity model
<code>to_smiles(molecule)</code>	Uses the OpenEye toolkit to convert a Molecule into a SMILES string.

`__init__(self, /, *args, **kwargs)`  
 Initialize self. See help(type(self)) for accurate signature.

## Methods

<code>__init__(self, /, *args, **kwargs)</code>	Initialize self.
<code>compute_partial_charges(self, molecule[, ...])</code>	Compute partial charges with OpenEye quacpac
<code>compute_partial_charges_am1bcc(self, molecule)</code>	Compute AM1BCC partial charges with OpenEye quacpac.
<code>compute_wiberg_bond_orders(self, molecule[, ...])</code>	Update and store list of bond orders this molecule.
<code>find_smarts_matches(self, molecule, smarts)</code>	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
<code>from_file(self, file_path, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file using this toolkit.
<code>from_file_obj(self, file_obj, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file-like object (an object with a “.read()” method using this toolkit.
<code>from_object(self, object[, ...])</code>	If given an OEMol (or OEMol-derived object), this function will load it into an openforcefield.topology.molecule Otherwise, it will return False.
<code>from_openeye(oemol[, low_undefined_stereo])</code>	al- Create a Molecule from an OpenEye molecule.
<code>from_smiles(self, smiles[, ...])</code>	Create a Molecule from a SMILES string using the OpenEye toolkit.
<code>generate_conformers(self, molecule[, ...])</code>	Generate molecule conformers using OpenEye Omega.
<code>is_available([oetools])</code>	Check if the given OpenEye toolkit components are available.
<code>requires_toolkit()</code>	
<code>to_file(self, molecule, file_path, file_format)</code>	Writes an OpenFF Molecule to a file-like object
<code>to_file_obj(self, molecule, file_obj, ...)</code>	Writes an OpenFF Molecule to a file-like object
<code>to_openeye(molecule[, aromaticity_model])</code>	Create an OpenEye molecule using the specified aromaticity model
<code>to_smiles(molecule)</code>	Uses the OpenEye toolkit to convert a Molecule into a SMILES string.

## Attributes

<code>toolkit_file_read_formats</code>	List of file formats that this toolkit can read.
<code>toolkit_file_write_formats</code>	List of file formats that this toolkit can write.
<code>toolkit_installation_instructions</code>	classmethod(function) -> method
<code>toolkit_name</code>	The name of the toolkit wrapped by this class.

**static is\_available**(*oetools*=('oechem', 'oequacpac', 'oeiupac', 'oeomega'))

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 given toolkits , False is returned.

#### Parameters

**oetools** [str or iterable of strings, optional, default=('oechem', 'oequacpac', 'oeiupac', 'oeomega')] Set of tools to check by their Python module name. Defaults to the complete set of tools supported by this function. Also accepts a single tool to check as a string instead of an iterable of length 1.

#### Returns

**all\_installed** [bool] True if all tools in *oetools* are installed and licensed, False otherwise

**from\_object**(*self, object, allow\_undefined\_stereo=False*)

If given an OEMol (or OEMol-derived object), this function will load it into an openforcefield.topology.molecule Otherwise, it will return False.

#### Parameters

**object** [A molecule-like object] An object to by type-checked.

**allow\_undefined\_stereo** [bool, default=False] Whether to accept molecules with undefined stereocenters. If False, an exception will be raised if a molecule with undefined stereochemistry is passed into this function.

#### Returns

**Molecule** An openforcefield.topology.molecule Molecule.

#### Raises

**NotImplementedError** If the object could not be converted into a Molecule.

**from\_file**(*self, file\_path, file\_format, allow\_undefined\_stereo=False*)

Return an openforcefield.topology.Molecule from a file using this toolkit.

#### Parameters

**file\_path** [str] The file to read the molecule from

**file\_format** [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit\_file\_read\_formats for details.

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

#### Returns

**molecules** [List[Molecule]] The list of Molecule objects in the file.

#### Raises

**GAFFAtomTypeWarning** If the loaded mol2 file possibly uses GAFF atom types, which are not supported.

### Examples

Load a mol2 file into an OpenFF Molecule object.

```
>>> from openforcefield.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**(*self*, *file\_obj*, *file\_format*, *allow\_undefined\_stereo=False*)

Return an openforcefield.topology.Molecule from a file-like object (an object with a “.read()” method using this toolkit.

#### Parameters

**file\_obj** [file-like object] The file-like object to read the molecule from

**file\_format** [str] Format specifier, usually file suffix (eg. ‘MOL2’, ‘SMI’) Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit\_file\_read\_formats for details.

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

#### Returns

**molecules** [List[Molecule]] The list of Molecule objects in the file object.

#### Raises

**GAFFAtomTypeWarning** If the loaded mol2 file possibly uses GAFF atom types, which are not supported.

**to\_file\_obj**(*self*, *molecule*, *file\_obj*, *file\_format*)

Writes an OpenFF Molecule to a file-like object

#### Parameters

**molecule** [an OpenFF Molecule] The molecule to write

**file\_obj** The file-like object to write to

**file\_format** The format for writing the molecule data

**to\_file**(*self*, *molecule*, *file\_path*, *file\_format*)

Writes an OpenFF Molecule to a file-like object

#### Parameters

**molecule** [an OpenFF Molecule] The molecule to write

**file\_path** The file path to write to.

**file\_format** The format for writing the molecule data

**static from\_openeye**(*oemol*, *allow\_undefined\_stereo=False*)

Create a Molecule from an OpenEye molecule. If the OpenEye molecule has implicit hydrogens, this function will make them explicit.

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

#### Parameters

**oemol** [openeye.oechem.OEMol] An OpenEye molecule

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

**Returns**

**molecule** [openforcefield.topology.Molecule] An openforcefield molecule

**Examples**

Create a Molecule from an OpenEye OEMol

```
>>> from openeye import oechem
>>> from openforcefield.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])
```

**static to\_openeye(molecule, aromaticity\_model='OEaroModel\_MDL')**

Create an OpenEye molecule using the specified aromaticity model

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

**Parameters**

**molecule** [openforcefield.topology.molecule.Molecule object]

The molecule to convert to an OEMol

**aromaticity\_model** [str, optional, default=DEFAULT\_AROMATICITY\_MODEL]

The aromaticity model to use

**Returns**

**oemol** [openeye.oechem.OEMol] An OpenEye molecule

**Examples**

Create an OpenEye molecule from a Molecule

```
>>> from openforcefield.topology import Molecule
>>> toolkit_wrapper = OpenEyeToolkitWrapper()
>>> molecule = Molecule.from_smiles('CC')
>>> oemol = toolkit_wrapper.to_openeye(molecule)
```

**static to\_smiles(molecule)**

Uses the OpenEye toolkit to convert a Molecule into a SMILES string.

**Parameters**

**molecule** [An openforcefield.topology.Molecule] The molecule to convert into a SMILES.

**Returns**

**smiles** [str] The SMILES of the input molecule.

**from\_smiles**(*self*, *smiles*, *hydrogens\_are\_explicit*=False, *allow\_undefined\_stereo*=False)  
Create a Molecule from a SMILES string using the OpenEye toolkit.

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

#### Parameters

**smiles** [str] The SMILES string to turn into a molecule

**hydrogens\_are\_explicit** [bool, default = False] If False, OE will perform hydrogen addition using OEAddExplicitHydrogens

**allow\_undefined\_stereo** [bool, default=False] Whether to accept SMILES with undefined stereochemistry. If False, an exception will be raised if a SMILES with undefined stereochemistry is passed into this function.

#### Returns

**molecule** [openforcefield.topology.Molecule] An openforcefield-style molecule.

**generate\_conformers**(*self*, *molecule*, *n\_conformers*=1, *clear\_existing*=True)  
Generate molecule conformers using OpenEye Omega.

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

**molecule** [a Molecule ] The molecule to generate conformers for.

**n\_conformers** [int, default=1] The maximum number of conformers to generate.

**clear\_existing** [bool, default=True] Whether to overwrite existing conformers for the molecule

**compute\_partial\_charges**(*self*, *molecule*, *quantum\_chemical\_method*='AM1-BCC', *partial\_charge\_method*='None')  
Compute partial charges with OpenEye quacpac

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

#### Parameters

**molecule** [Molecule] Molecule for which partial charges are to be computed

**charge\_model** [str, optional, default=None] The charge model to use. One of ['noop', 'mmff', 'mmff94', 'am1bcc', 'am1bccnosymspt', 'amber', 'amberff94', 'am1bccelf10'] If None, 'am1bcc' will be used.

#### Returns

**charges** [numpy.array of shape (natoms) of type float] The partial charges

**compute\_partial\_charges\_am1bcc**(*self*, *molecule*)  
Compute AM1BCC partial charges with OpenEye quacpac. This function will attempt to use the OEAM1BCCSELF10 charge generation method, but may print a warning and fall back to normal OEAM1BCC if an error is encountered. This error is known to occur with some carboxylic acids, and is under investigation by OpenEye.

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

#### Parameters

**molecule** [Molecule] Molecule for which partial charges are to be computed

#### Returns

**charges** [numpy.array of shape (natoms) of type float] The partial charges

**compute\_wiberg\_bond\_orders**(*self*, *molecule*, *charge\_model=None*)

Update and store list of bond orders this molecule. Can be used for initialization of bondorders list, or for updating bond orders in the list.

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

#### Parameters

**molecule** [openforcefield.topology.molecule Molecule] The molecule to assign wiberg bond orders to

**charge\_model** [str, optional, default=None] The charge model to use. One of ['am1', 'pm3']. If None, 'am1' will be used.

**find\_smarts\_matches**(*self*, *molecule*, *smarts*, *aromaticity\_model='OEAroModel\_MDL'*)

Find all SMARTS matches for the specified molecule, using the specified aromaticity model.

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

#### Parameters

**molecule** [openforcefield.topology.Molecule] The molecule for which all specified SMARTS matches are to be located

**smarts** [str] SMARTS string with optional SMIRKS-style atom tagging

**aromaticity\_model** [str, optional, default='OEAroModel\_MDL'] Aromaticity model to use during matching

**.. note ::** Currently, the only supported “aromaticity\_model“ is “OEAroModel\_MDL“

**property toolkit\_file\_read\_formats**

List of file formats that this toolkit can read.

**property toolkit\_file\_write\_formats**

List of file formats that this toolkit can write.

**property toolkit\_installation\_instructions**

classmethod(function) -> method

Convert a function to be a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C: @classmethod def f(cls, arg1, arg2, ...):
```

```
...
```

It can be called either on the class (e.g. `C.f()`) or on an instance (e.g. `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see the `staticmethod` builtin.

**property** `toolkit_name`

The name of the toolkit wrapped by this class.

## openforcefield.utils.toolkits.RDKitToolkitWrapper

```
class openforcefield.utils.toolkits.RDKitToolkitWrapper
```

RDKit toolkit wrapper

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

### Attributes

`toolkit_file_read_formats` List of file formats that this toolkit can read.

`toolkit_file_write_formats` List of file formats that this toolkit can write.

`toolkit_installation_instructions` classmethod(function) -> method

`toolkit_name` The name of the toolkit wrapped by this class.

### Methods

<code>find_smarts_matches(self, molecule, smarts)</code>	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
<code>from_file(self, file_path, file_format[, ...])</code>	Create an <code>openforcefield.topology.Molecule</code> from a file using this toolkit.
<code>from_file_obj(self, file_obj, file_format[, ...])</code>	Return an <code>openforcefield.topology.Molecule</code> from a file-like object (an object with a “ <code>read()</code> ” method using this toolkit.
<code>from_object(self, object[, ...])</code>	If given an <code>rdchem.Mol</code> (or <code>rdchem.Mol</code> -derived object), this function will load it into an <code>openforcefield.topology.molecule</code> .
<code>from_rdkit(self, rdmol[, allow_undefined_stereo])</code>	Create a Molecule from an RDKit molecule.
<code>from_smiles(self, smiles[, ...])</code>	Create a Molecule from a SMILES string using the RDKit toolkit.
<code>generate_conformers(self, molecule[, ...])</code>	Generate molecule conformers using RDKit.
<code>is_available()</code>	Check whether the RDKit toolkit can be imported
<code>requires_toolkit()</code>	
<code>to_file(self, molecule, file_path, file_format)</code>	Writes an OpenFF Molecule to a file-like object
<code>to_file_obj(self, molecule, file_obj, ...)</code>	Writes an OpenFF Molecule to a file-like object

Continued on next page



Table 125 – continued from previous page

<code>to_rdkit(molecule[, aromaticity_model])</code>	Create an RDKit molecule
<code>to_smiles(molecule)</code>	Uses the RDKit toolkit to convert a Molecule into a SMILES string.

`__init__(self, /, *args, **kwargs)`  
 Initialize self. See help(type(self)) for accurate signature.

## Methods

<code>__init__(self, /, *args, **kwargs)</code>	Initialize self.
<code>find_smarts_matches(self, molecule, smarts)</code>	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
<code>from_file(self, file_path, file_format[, ...])</code>	Create an openforcefield.topology.Molecule from a file using this toolkit.
<code>from_file_obj(self, file_obj, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file-like object (an object with a “ <code>.read()</code> ” method using this toolkit.
<code>from_object(self, object[, ...])</code>	If given an rdchem.Mol (or rdchem.Mol-derived object), this function will load it into an openforcefield.topology.molecule.
<code>from_rdkit(self, rdmol[, allow_undefined_stereo])</code>	Create a Molecule from an RDKit molecule.
<code>from_smiles(self, smiles[, ...])</code>	Create a Molecule from a SMILES string using the RDKit toolkit.
<code>generate_conformers(self, molecule[, ...])</code>	Generate molecule conformers using RDKit.
<code>is_available()</code>	Check whether the RDKit toolkit can be imported
<code>requires_toolkit()</code>	
<code>to_file(self, molecule, file_path, file_format)</code>	Writes an OpenFF Molecule to a file-like object
<code>to_file_obj(self, molecule, file_obj, ...)</code>	Writes an OpenFF Molecule to a file-like object
<code>to_rdkit(molecule[, aromaticity_model])</code>	Create an RDKit molecule
<code>to_smiles(molecule)</code>	Uses the RDKit toolkit to convert a Molecule into a SMILES string.

## Attributes

<code>toolkit_file_read_formats</code>	List of file formats that this toolkit can read.
<code>toolkit_file_write_formats</code>	List of file formats that this toolkit can write.
<code>toolkit_installation_instructions</code>	classmethod(function) -> method
<code>toolkit_name</code>	The name of the toolkit wrapped by this class.

**static `is_available()`**  
 Check whether the RDKit toolkit can be imported

## Returns

**`is_installed`** [bool] True if RDKit is installed, False otherwise.

**`from_object(self, object, allow_undefined_stereo=False)`**  
 If given an rdchem.Mol (or rdchem.Mol-derived object), this function will load it into an openforcefield.topology.molecule. Otherwise, it will return False.

**Parameters**

**object** [A rdchem.Mol-derived object] An object to be type-checked and converted into a Molecule, if possible.

**allow\_undefined\_stereo** [bool, default=False] Whether to accept molecules with undefined stereocenters. If False, an exception will be raised if a molecule with undefined stereochemistry is passed into this function.

**Returns**

**Molecule or False** An openforcefield.topology.molecule Molecule.

**Raises**

**NotImplementedError** If the object could not be converted into a Molecule.

**from\_file**(*self*, *file\_path*, *file\_format*, *allow\_undefined\_stereo*=False)

Create an openforcefield.topology.Molecule from a file using this toolkit.

**Parameters**

**file\_path** [str] The file to read the molecule from

**file\_format** [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit\_file\_read\_formats for details.

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

**Returns**

**molecules** [iterable of Molecules] a list of Molecule objects is returned.

**from\_file\_obj**(*self*, *file\_obj*, *file\_format*, *allow\_undefined\_stereo*=False)

Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.

<b>Warning:</b> This API is experimental and subject to change.
---

**Parameters**

**file\_obj** [file-like object] The file-like object to read the molecule from

**file\_format** [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit\_file\_read\_formats for details.

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

**Returns**

**molecules** [Molecule or list of Molecules] a list of Molecule objects is returned.

**to\_file\_obj**(*self*, *molecule*, *file\_obj*, *file\_format*)

Writes an OpenFF Molecule to a file-like object

**Parameters**

**molecule** [an OpenFF Molecule] The molecule to write

**file\_obj** The file-like object to write to

**file\_format** The format for writing the molecule data

**to\_file**(*self*, *molecule*, *file\_path*, *file\_format*)

Writes an OpenFF Molecule to a file-like object

#### Parameters

**molecule** [an OpenFF Molecule] The molecule to write

**file\_path** The file path to write to

**file\_format** The format for writing the molecule data

#### Returns

—

**classmethod to\_smiles**(*molecule*)

Uses the RDKit toolkit to convert a Molecule into a SMILES string.

#### Parameters

**molecule** [An `openforcefield.topology.Molecule`] The molecule to convert into a SMILES.

#### Returns

**smiles** [str] The SMILES of the input molecule.

**from\_smiles**(*self*, *smiles*, *hydrogens\_are\_explicit=False*, *allow\_undefined\_stereo=False*)

Create a Molecule from a SMILES string using the RDKit toolkit.

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

#### Parameters

**smiles** [str] The SMILES string to turn into a molecule

**hydrogens\_are\_explicit** [bool, default=False] If False, RDKit will perform hydrogen addition using `Chem.AddHs`

**allow\_undefined\_stereo** [bool, default=False] Whether to accept SMILES with undefined stereochemistry. If False, an exception will be raised if a SMILES with undefined stereochemistry is passed into this function.

#### Returns

**molecule** [`openforcefield.topology.Molecule`] An openforcefield-style molecule.

**generate\_conformers**(*self*, *molecule*, *n\_conformers=1*, *clear\_existing=True*)

Generate molecule conformers using RDKit.

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

**molecule** [a `Molecule`] The molecule to generate conformers for.

**n\_conformers** [int, default=1] Maximum number of conformers to generate.

**clear\_existing** [bool, default=True] Whether to overwrite existing conformers for the molecule.

**from\_rdkit**(*self*, *rdmol*, *allow\_undefined\_stereo=False*)

Create a Molecule from an RDKit molecule.

Requires the RDKit to be installed.

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

#### Parameters

**rdmol** [rdkit.RDMol] An RDKit molecule

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if rdmol contains undefined stereochemistry.

#### Returns

**molecule** [openforcefield.Molecule] An openforcefield molecule

#### Examples

Create a molecule from an RDKit molecule

```
>>> from rdkit import Chem
>>> from openforcefield.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)
```

**classmethod to\_rdkit**(*molecule*, *aromaticity\_model='OEAroModel\_MDL'*)

Create an RDKit molecule

Requires the RDKit to be installed.

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

#### Parameters

**aromaticity\_model** [str, optional, default=DEFAULT\_AROMATICITY\_MODEL] The aromaticity model to use

#### Returns

**rdmol** [rdkit.RDMol] An RDKit molecule

#### Examples

Convert a molecule to RDKit

```
>>> from openforcefield.topology import Molecule
>>> ethanol = Molecule.from_smiles('CCO')
>>> rdmol = ethanol.to_rdkit()
```

**find\_smarts\_matches**(*self*, *molecule*, *smarts*, *aromaticity\_model*=*'OEAroModel\_MDL'*)  
Find all SMARTS matches for the specified molecule, using the specified aromaticity model.

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

#### Parameters

**molecule** [openforcefield.topology.Molecule] The molecule for which all specified SMARTS matches are to be located

**smarts** [str] SMARTS string with optional SMIRKS-style atom tagging

**aromaticity\_model** [str, optional, default=*'OEAroModel\_MDL'*] Aromaticity model to use during matching

**.. note ::** Currently, the only supported “aromaticity\_model” is “OEAroModel\_MDL”

#### property toolkit\_file\_read\_formats

List of file formats that this toolkit can read.

#### property toolkit\_file\_write\_formats

List of file formats that this toolkit can write.

#### property toolkit\_installation\_instructions

classmethod(function) -> method

Convert a function to be a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C: @classmethod def f(cls, arg1, arg2, ...):
```

```
...
```

It can be called either on the class (e.g. *C.f()*) or on an instance (e.g. *C().f()*). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see the *staticmethod* builtin.

#### property toolkit\_name

The name of the toolkit wrapped by this class.

### openforcefield.utils.toolkits.AmberToolsToolkitWrapper

```
class openforcefield.utils.toolkits.AmberToolsToolkitWrapper
    AmberTools toolkit wrapper
```

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

#### Attributes

**toolkit\_file\_read\_formats** List of file formats that this toolkit can read.

**toolkit\_file\_write\_formats** List of file formats that this toolkit can write.

`toolkit_installation_instructions` classmethod(function) -> method

`toolkit_name` The name of the toolkit wrapped by this class.

## Methods

<code>compute_partial_charges(self, molecule[, ...])</code>	Compute partial charges with AmberTools using antechamber/sqm
<code>compute_partial_charges_am1bcc(self, molecule)</code>	Compute partial charges with AmberTools using antechamber/sqm.
<code>from_file(self, file_path, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file using this toolkit.
<code>from_file_obj(self, file_obj, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file-like object (an object with a “.read()” method using this
<code>is_available()</code>	Check whether the AmberTools toolkit is installed
<code>requires_toolkit()</code>	

`__init__(self)`

Initialize self. See help(type(self)) for accurate signature.

## Methods

<code>__init__(self)</code>	Initialize self.
<code>compute_partial_charges(self, molecule[, ...])</code>	Compute partial charges with AmberTools using antechamber/sqm
<code>compute_partial_charges_am1bcc(self, molecule)</code>	Compute partial charges with AmberTools using antechamber/sqm.
<code>from_file(self, file_path, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file using this toolkit.
<code>from_file_obj(self, file_obj, file_format[, ...])</code>	Return an openforcefield.topology.Molecule from a file-like object (an object with a “.read()” method using this
<code>is_available()</code>	Check whether the AmberTools toolkit is installed
<code>requires_toolkit()</code>	

## Attributes

<code>toolkit_file_read_formats</code>	List of file formats that this toolkit can read.
<code>toolkit_file_write_formats</code>	List of file formats that this toolkit can write.
<code>toolkit_installation_instructions</code>	classmethod(function) -> method
<code>toolkit_name</code>	The name of the toolkit wrapped by this class.

**static** `is_available()`

Check whether the AmberTools toolkit is installed

## Returns

**is\_installed** [bool] True if AmberTools is installed, False otherwise.

**compute\_partial\_charges**(*self*, *molecule*, *charge\_model*=None)  
 Compute partial charges with AmberTools using antechamber/sqm

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

#### Parameters

**molecule** [Molecule] Molecule for which partial charges are to be computed

**charge\_model** [str, optional, default=None] The charge model to use. One of ['gas', 'mul', 'bcc']. If None, 'bcc' will be used.

#### Raises

**ValueError** if the requested charge method could not be handled

#### Notes

Currently only sdf file supported as input and mol2 as output <https://github.com/choderalab/openmoltools/blob/master/openmoltools/packmol.py>

**compute\_partial\_charges\_am1bcc**(*self*, *molecule*)  
 Compute partial charges with AmberTools using antechamber/sqm. This will calculate AM1-BCC charges on the first conformer only.

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

#### Parameters

**molecule** [Molecule] Molecule for which partial charges are to be computed

#### Raises

**ValueError** if the requested charge method could not be handled

**from\_file**(*self*, *file\_path*, *file\_format*, *allow\_undefined\_stereo*=False)  
 Return an openforcefield.topology.Molecule from a file using this toolkit.

#### Parameters

**file\_path** [str] The file to read the molecule from

**file\_format** [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit\_file\_read\_formats for details.

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if any molecules contain undefined stereochemistry.

#### Returns

**molecules** [Molecule or list of Molecules] a list of Molecule objects is returned.

**from\_file\_obj**(*self*, *file\_obj*, *file\_format*, *allow\_undefined\_stereo*=False)

Return an `openforcefield.topology.Molecule` from a file-like object (an object with a `“read()”` method using toolkit).

#### Parameters

**file\_obj** [file-like object] The file-like object to read the molecule from

**file\_format** [str] Format specifier, usually file suffix (eg. ‘MOL2’, ‘SMI’) Note that not all toolkits support all formats. Check `ToolkitWrapper.toolkit_file_read_formats` for details.

**allow\_undefined\_stereo** [bool, default=False] If false, raises an exception if any molecules contain undefined stereochemistry. If false, the function skips loading the molecule.

#### Returns

**molecules** [Molecule or list of Molecules] a list of Molecule objects is returned.

#### property `toolkit_file_read_formats`

List of file formats that this toolkit can read.

#### property `toolkit_file_write_formats`

List of file formats that this toolkit can write.

#### property `toolkit_installation_instructions`

`classmethod(function) -> method`

Convert a function to be a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C: @classmethod def f(cls, arg1, arg2, ...):
```

```
...
```

It can be called either on the class (e.g. `C.f()`) or on an instance (e.g. `CO.f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see the `staticmethod` builtin.

#### property `toolkit_name`

The name of the toolkit wrapped by this class.

## 2.3.2 Serialization support

---

`Serializable`

Mix-in to add serialization and deserialization support via JSON, YAML, BSON, TOML, MessagePack, and XML.

---

### `openforcefield.utils.serialization.Serializable`

#### **class** `openforcefield.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/deserialization 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 openforcefield.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()
```

## Methods

<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.
<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

<code>from_dict</code>	
<code>to_dict</code>	

`__init__(self, /, *args, **kwargs)`  
Initialize self. See `help(type(self))` for accurate signature.

## Methods

<code>__init__(self, /, *args, **kwargs)</code>	Initialize self.
<code>from_bson(serialized)</code>	Instantiate an object from a BSON serialized representation.
<code>from_dict(d)</code>	
<code>from_json(serialized)</code>	Instantiate an object from a JSON serialized representation.

Continued on next page

Table 133 – continued from previous page

<code>from_messagepack(serialized)</code>	Instantiate an object from a MessagePack serialized representation.
<code>from_pickle(serialized)</code>	Instantiate an object from a pickle serialized representation.
<code>from_toml(serialized)</code>	Instantiate an object from a TOML serialized representation.
<code>from_xml(serialized)</code>	Instantiate an object from an XML serialized representation.
<code>from_yaml(serialized)</code>	Instantiate from a YAML serialized representation.
<code>to_bson(self)</code>	Return a BSON serialized representation.
<code>to_dict(self)</code>	
<code>to_json(self[, indent])</code>	Return a JSON serialized representation.
<code>to_messagepack(self)</code>	Return a MessagePack representation.
<code>to_pickle(self)</code>	Return a pickle serialized representation.
<code>to_toml(self)</code>	Return a TOML serialized representation.
<code>to_xml(self[, indent])</code>	Return an XML representation.
<code>to_yaml(self)</code>	Return a YAML serialized representation.

**`to_json(self, indent=None)`**

Return a JSON serialized representation.

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

#### Parameters

**`indent`** [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

#### Returns

**`serialized`** [str] A JSON serialized representation of the object

**`classmethod from_json(serialized)`**

Instantiate an object from a JSON serialized representation.

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

#### Parameters

**`serialized`** [str] A JSON serialized representation of the object

#### Returns

**`instance`** [cls] An instantiated object

**`to_bson(self)`**

Return a BSON serialized representation.

Specification: <http://bsonspec.org/>

#### Returns

**`serialized`** [bytes] A BSON serialized representation of the object

**`classmethod from_bson(serialized)`**

Instantiate an object from a BSON serialized representation.

Specification: <http://bsonspec.org/>

#### Parameters

**serialized** [bytes] A BSON serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**to\_toml**(*self*)

Return a TOML serialized representation.

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

**Returns**

**serialized** [str] A TOML serialized representation of the object

**classmethod from\_toml**(*serialized*)

Instantiate an object from a TOML serialized representation.

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

**Parameters**

**serialized** [str] A TOML serialized representation of the object

**Returns**

**instance** [cls] An instantiated object

**to\_yaml**(*self*)

Return a YAML serialized representation.

Specification: <http://yaml.org/>

**Returns**

**serialized** [str] A YAML serialized representation of the object

**classmethod from\_yaml**(*serialized*)

Instantiate from a YAML serialized representation.

Specification: <http://yaml.org/>

**Parameters**

**serialized** [str] A YAML serialized representation of the object

**Returns**

**instance** [cls] Instantiated object

**to\_messagepack**(*self*)

Return a MessagePack representation.

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

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation of the object

**classmethod from\_messagepack**(*serialized*)

Instantiate an object from a MessagePack serialized representation.

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

**Parameters**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation

**Returns**

**instance** [cls] Instantiated object.

**to\_xml**(*self*, *indent*=2)

Return an XML representation.

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

**Parameters**

**indent** [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

**Returns**

**serialized** [bytes] A MessagePack-encoded bytes serialized representation.

**classmethod from\_xml**(*serialized*)

Instantiate an object from an XML serialized representation.

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

**Parameters**

**serialized** [bytes] An XML serialized representation

**Returns**

**instance** [cls] Instantiated object.

**to\_pickle**(*self*)

Return a pickle serialized representation.

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

**Returns**

**serialized** [str] A pickled representation of the object

**classmethod from\_pickle**(*serialized*)

Instantiate an object from a pickle serialized representation.

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

**Parameters**

**serialized** [str] A pickled representation of the object

**Returns**

**instance** [cls] An instantiated object

### 2.3.3 Miscellaneous utilities

Miscellaneous utility functions.

<code>inherit_docstrings</code>	Inherit docstrings from parent class
<code>all_subclasses</code>	Recursively retrieve all subclasses of the specified class
<code>temporary_cd</code>	Context to temporary change the working directory.
<code>temporary_directory</code>	Context for safe creation of temporary directories.
<code>get_data_file_path</code>	Get the full path to one of the reference files in test-systems.
<code>convert_0_1_smirnoff_to_0_2</code>	Convert an 0.1-compliant SMIRNOFF dict to an 0.2-compliant one.
<code>convert_0_2_smirnoff_to_0_3</code>	Convert an 0.2-compliant SMIRNOFF dict to an 0.3-compliant one.

### `openforcefield.utils.utils.inherit_docstrings`

`openforcefield.utils.utils.inherit_docstrings(cls)`  
Inherit docstrings from parent class

### `openforcefield.utils.utils.all_subclasses`

`openforcefield.utils.utils.all_subclasses(cls)`  
Recursively retrieve all subclasses of the specified class

### `openforcefield.utils.utils.temporary_cd`

`openforcefield.utils.utils.temporary_cd(dir_path)`  
Context to temporary change the working directory.

#### Parameters

**dir\_path** [str] The directory path to enter within the context

#### Examples

```
>>> dir_path = '/tmp'
>>> with temporary_cd(dir_path):
...     pass # do something in dir_path
```

### `openforcefield.utils.utils.temporary_directory`

`openforcefield.utils.utils.temporary_directory()`  
Context for safe creation of temporary directories.

### `openforcefield.utils.utils.get_data_file_path`

`openforcefield.utils.utils.get_data_file_path(relative_path)`  
Get the full path to one of the reference files in testsystems. In the source distribution, these files are in `openforcefield/data/`, but on installation, they're moved to somewhere in the user's python site-packages directory. Parameters ——— name : str  
Name of the file to load (with respect to the repex folder).

### openforcefield.utils.convert\_0\_1\_smirnoff\_to\_0\_2

openforcefield.utils.convert\_0\_1\_smirnoff\_to\_0\_2(*smirnoff\_data\_0\_1*)

Convert an 0.1-compliant SMIRNOFF dict to an 0.2-compliant one. This involves renaming several tags, adding Electrostatics and ToolkitAM1BCC tags, and separating improper torsions into their own section.

#### Parameters

**smirnoff\_data\_0\_1** [dict] Hierarchical dict representing a SMIRNOFF data structure according to the 0.1 spec

#### Returns

**smirnoff\_data\_0\_2** Hierarchical dict representing a SMIRNOFF data structure according to the 0.2 spec

### openforcefield.utils.convert\_0\_2\_smirnoff\_to\_0\_3

openforcefield.utils.convert\_0\_2\_smirnoff\_to\_0\_3(*smirnoff\_data\_0\_2*)

Convert an 0.2-compliant SMIRNOFF dict to an 0.3-compliant one. This involves removing units from header tags and adding them to attributes of child elements. It also requires converting ProperTorsions and ImproperTorsions potentials from “charmm” to “fourier”.

#### Parameters

**smirnoff\_data\_0\_2** [dict] Hierarchical dict representing a SMIRNOFF data structure according to the 0.2 spec

#### Returns

**smirnoff\_data\_0\_3** Hierarchical dict representing a SMIRNOFF data structure according to the 0.3 spec

## 2.3.4 Structure tools

Tools for manipulating molecules and structures

**Warning:** These methods are deprecated and will be removed and replaced with integrated API methods. We recommend that no new code makes use of these functions.

---

`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.

---

### openforcefield.utils.structure.get\_molecule\_parameterIDs

openforcefield.utils.structure.get\_molecule\_parameterIDs(*molecules*, *forcefield*)

Process a list of molecules with a specified SMIRNOFF ffxml file and determine which parameters are used by which molecules, returning collated results.

#### Parameters

**molecules** [list of openforcefield.topology.Molecule] List of molecules (with explicit hydrogens) to parse

**forcefield** [openforcefield.typing.engines.smirnoff.ForceField] The ForceField to apply

#### Returns

**parameters\_by\_molecule** [dict] Parameter IDs used in each molecule, keyed by isomeric SMILES generated from provided OEMols. Each entry in the dict is a list which does not necessarily have unique entries; i.e. parameter IDs which are used more than once will occur multiple times.

**parameters\_by\_ID** [dict] Molecules in which each parameter ID occur, keyed by parameter ID. Each entry in the dict is a set of isomeric SMILES for molecules in which that parameter occurs. No frequency information is stored.



## INDEX

### Symbols

`__init__()` (*openforcefield.topology.Atom* method), 105  
`__init__()` (*openforcefield.topology.Bond* method), 111  
`__init__()` (*openforcefield.topology.FrozenMolecule* method), 42  
`__init__()` (*openforcefield.topology.Molecule* method), 60  
`__init__()` (*openforcefield.topology.Particle* method), 100  
`__init__()` (*openforcefield.topology.Topology* method), 82  
`__init__()` (*openforcefield.topology.TopologyAtom* method), 122  
`__init__()` (*openforcefield.topology.TopologyBond* method), 127  
`__init__()` (*openforcefield.topology.TopologyMolecule* method), 95  
`__init__()` (*openforcefield.topology.TopologyVirtualSite* method), 132  
`__init__()` (*openforcefield.topology.VirtualSite* method), 116  
`__init__()` (*openforcefield.typing.chemistry.ChemicalEnvironment* method), 139  
`__init__()` (*openforcefield.typing.engines.smirnoff.forcefield.ForceField* method), 148  
`__init__()` (*openforcefield.typing.engines.smirnoff.io.ParameterIOHandler* method), 375  
`__init__()` (*openforcefield.typing.engines.smirnoff.io.XMLParameterIOHandler* method), 377  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler* method), 250  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler.AngleType* method), 176  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.BondHandler* method), 233  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.BondHandler.BondType* method), 166  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler* method), 342  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler* method), 363  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler.GBSAType* method), 217  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler* method), 285  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler.ImproperTorsionType* method), 196  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.IndexedParameterAttribute* method), 382  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.ParameterAttribute* method), 380  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler* method), 223  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.ParameterList* method), 219  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.ParameterType* method), 156  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler* method), 267  
`__init__()` (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler.ProperTorsionType* method), 186

`__init__()` (`openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler`  
`field.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler` method), 343  
`__init__()` (`openforcefield.typing.engines.smirnoff.parameters.vdWHandler` method), 315  
`__init__()` (`openforcefield.typing.engines.smirnoff.parameters.vdWHandler.vdWType` method), 207  
`__init__()` (`openforcefield.utils.serialization.Serializable` method), 406  
`__init__()` (`openforcefield.utils.toolkits.AmberToolsToolkitWrapper` method), 402  
`__init__()` (`openforcefield.utils.toolkits.OpenEyeToolkitWrapper` method), 390  
`__init__()` (`openforcefield.utils.toolkits.RDKitToolkitWrapper` method), 397  
`__init__()` (`openforcefield.utils.toolkits.ToolkitRegistry` method), 384  
`__init__()` (`openforcefield.utils.toolkits.ToolkitWrapper` method), 387  
**A**  
`add_atom()` (`openforcefield.topology.Molecule` method), 76  
`add_bond()` (`openforcefield.topology.Atom` method), 107  
`add_bond()` (`openforcefield.topology.Molecule` method), 78  
`add_bond_charge_virtual_site()` (`openforcefield.topology.Molecule` method), 76  
`add_conformer()` (`openforcefield.topology.Molecule` method), 79  
`add_constraint()` (`openforcefield.topology.Topology` method), 93  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.AngleHandler` method), 260  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.AngleHandler.AngleType` method), 176, 259  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.BondHandler` method), 244  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.BondHandler.BondType` method), 166, 243  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler` method), 343  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.GBSAHandler` method), 373  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.GBSAHandler.GBSAType` method), 218, 372  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler` method), 296  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler.ImproperTorsionType` method), 197, 295  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.ParameterHandler` method), 226  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.ParameterType` method), 157  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler` method), 278  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler.ProperTorsionType` method), 186, 277  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler` method), 349  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.vdWHandler` method), 327  
`add_cosmetic_attribute()` (`openforcefield.typing.engines.smirnoff.parameters.vdWHandler.vdWType` method), 208, 326  
`add_divalent_lone_pair_virtual_site()` (`openforcefield.topology.Molecule` method), 77  
`add_molecule()` (`openforcefield.topology.Topology` method), 91  
`add_monovalent_lone_pair_virtual_site()` (`openforcefield.topology.Molecule` method), 77  
`add_parameter()` (`openforcefield.typing.engines.smirnoff.parameters.AngleHandler` method), 260  
`add_parameter()` (`openforcefield.typing.engines.smirnoff.parameters.BondHandler` method), 244  
`add_parameter()` (`openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler` method), 344  
`add_parameter()` (`openforcefield.typing.engines.smirnoff.parameters.GBSAHandler` method), 373  
`add_parameter()` (`openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler` method), 296  
`add_parameter()` (`openforcefield.typing.engines.smirnoff.parameters.ParameterHandler` method), 226  
`add_parameter()` (`openforcefield.typing.engines.smirnoff.parameters.ParameterType` method), 157  
`add_parameter()` (`openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler` method), 278  
`add_parameter()` (`openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler.ProperTorsionType` method), 186, 277  
`add_parameter()` (`openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler` method), 349  
`add_parameter()` (`openforcefield.typing.engines.smirnoff.parameters.vdWHandler` method), 327  
`add_parameter()` (`openforcefield.typing.engines.smirnoff.parameters.vdWHandler.vdWType` method), 208, 326

`field.typing.engines.smirnoff.parameters.ImproperTorsionHandler` (openforcefield.topology.TopologyMolecule property), 98  
`method`), 296  
`add_parameter()` (openforce- `AngleType` (class in openforce-  
`field.typing.engines.smirnoff.parameters.ParameterHandler`), 167  
`method`), 225  
`add_parameter()` (openforce- `append()` (openforcefield.typing.engines.smirnoff.parameters.Parameters  
`field.typing.engines.smirnoff.parameters.ProperTorsionHandler`), 220  
`method`), 278  
`add_parameter()` (openforce- `aromaticity_model()` (openforce-  
`field.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler`), 86  
`method`), 349  
`add_parameter()` (openforce- `assign_charge_from_molecules()` (openforce-  
`field.typing.engines.smirnoff.parameters.vdWHandler`), 349  
`method`), 327  
`add_particle()` (openforcefield.topology.Topology `assign_parameters()` (openforce-  
`method`), 91  
`add_toolkit()` (openforce- `field.typing.engines.smirnoff.parameters.AngleHandler`  
`field.utils.toolkits.ToolkitRegistry` `method`), 385  
`add_trivalent_lone_pair_virtual_site()` (open-  
`forcefield.topology.Molecule` `method`), 78  
`add_virtual_site()` (openforcefield.topology.Atom  
`method`), 107  
`addANDtype()` (openforce- `assign_parameters()` (openforce-  
`field.typing.chemistry.ChemicalEnvironment.Atom` `field.typing.engines.smirnoff.parameters.GBSAHandler`  
`method`), 141  
`addANDtype()` (openforce- `assign_parameters()` (openforce-  
`field.typing.chemistry.ChemicalEnvironment.Bond` `field.typing.engines.smirnoff.parameters.ImproperTorsionHandl`  
`method`), 142  
`addAtom()` (openforce- `assign_parameters()` (openforce-  
`field.typing.chemistry.ChemicalEnvironment` `field.typing.engines.smirnoff.parameters.ParameterHandler`  
`method`), 144  
`addORtype()` (openforce- `assign_parameters()` (openforce-  
`field.typing.chemistry.ChemicalEnvironment.Atom` `field.typing.engines.smirnoff.parameters.ProperTorsionHandler`  
`method`), 141  
`addORtype()` (openforce- `assign_parameters()` (openforce-  
`field.typing.chemistry.ChemicalEnvironment.Bond` `field.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandl`  
`method`), 142  
`all_subclasses()` (in module openforce- `assign_parameters()` (openforce-  
`field.utils.utils`), 410  
`AmberToolsToolkitWrapper` (class in openforce-  
`field.utils.toolkits`), 401  
`AngleHandler` (class in openforce-  
`field.typing.engines.smirnoff.parameters`),  
246  
`AngleHandler.AngleType` (class in openforce-  
`field.typing.engines.smirnoff.parameters`),  
251  
`angles()` (openforcefield.topology.FrozenMolecule  
property), 49  
`angles()` (openforcefield.topology.Molecule property),  
64  
`angles()` (openforcefield.topology.Topology property),  
88  
`asSMARTS()` (openforce-  
`field.typing.chemistry.ChemicalEnvironment.Atom`  
`method`), 141  
`asSMARTS()` (openforce-  
`field.typing.chemistry.ChemicalEnvironment.Bond`  
`method`), 142  
`asSMIRKS()` (openforce-  
`field.typing.chemistry.ChemicalEnvironment`  
`method`), 143  
`asSMIRKS()` (openforce-  
`field.typing.chemistry.ChemicalEnvironment.Atom`  
`method`), 141  
`asSMIRKS()` (openforce-

- field.typing.chemistry.ChemicalEnvironment.BondType* (class in *openforcefield.typing.engines.smirnoff.parameters*), 226
- BondType* (class in *openforcefield.typing.engines.smirnoff.parameters*), 235
- bonds()* (*openforcefield.topology.Atom* property), 107
- bonds()* (*openforcefield.topology.Molecule* property), 64
- bonds()* (*openforcefield.topology.TopologyMolecule* property), 97
- BondType* (class in *openforcefield.typing.engines.smirnoff.parameters*), 157
- box\_vectors()* (*openforcefield.topology.Topology* property), 86
- ## C
- call()* (*openforcefield.utils.toolkits.ToolkitRegistry* method), 386
- charge\_increments()* (*openforcefield.topology.VirtualSite* property), 118
- charge\_model()* (*openforcefield.topology.Topology* property), 86
- check\_handler\_compatibility()* (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler* method), 260
- check\_handler\_compatibility()* (*openforcefield.typing.engines.smirnoff.parameters.BondHandler* method), 244
- check\_handler\_compatibility()* (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler* method), 343
- check\_handler\_compatibility()* (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler* method), 375
- check\_handler\_compatibility()* (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler* method), 296
- check\_handler\_compatibility()* (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler* method), 224
- check\_handler\_compatibility()* (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler* method), 278
- check\_handler\_compatibility()* (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler* method), 349
- check\_handler\_compatibility()* (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler* method), 327
- chemical\_environment\_matches()* (*openforcefield.topology.FrozenMolecule* method), 50
- chemical\_environment\_matches()* (*openforcefield.topology.Molecule* method), 64
- chemical\_environment\_matches()* (*openforcefield.topology.Topology* method), 88
- ## B
- Bond* (class in *openforcefield.topology*), 110
- bond()* (*openforcefield.topology.Topology* method), 90
- bond()* (*openforcefield.topology.TopologyBond* property), 128
- bond()* (*openforcefield.topology.TopologyMolecule* method), 97
- bond\_order()* (*openforcefield.topology.TopologyBond* property), 128
- bond\_start\_topology\_index()* (*openforcefield.topology.TopologyMolecule* property), 98
- bonded\_atoms()* (*openforcefield.topology.Atom* property), 107
- BondHandler* (class in *openforcefield.typing.engines.smirnoff.parameters*), 226
- BondHandler.BondType* (class in *openforcefield.typing.engines.smirnoff.parameters*), 235
- bonds()* (*openforcefield.topology.Atom* property), 107
- bonds()* (*openforcefield.topology.FrozenMolecule* property), 49
- bonds()* (*openforcefield.topology.Molecule* property), 64
- bonds()* (*openforcefield.topology.TopologyBond* property), 129
- bonds()* (*openforcefield.topology.TopologyMolecule* property), 97
- bonds()* (*openforcefield.topology.TopologyVirtualSite* property), 134
- bonds()* (*openforcefield.topology.VirtualSite* property), 118
- author()* (*openforcefield.typing.engines.smirnoff.forcefield.ForceField* method), 150



ChemicalEnvironment (class in *openforcefield.typing.chemistry*), 138  
 ChemicalEnvironment.Atom (class in *openforcefield.typing.chemistry*), 140  
 ChemicalEnvironment.Bond (class in *openforcefield.typing.chemistry*), 141  
 clear() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList* method), 221  
 compute\_partial\_charges() (*openforcefield.topology.FrozenMolecule* method), 47  
 compute\_partial\_charges() (*openforcefield.topology.Molecule* method), 64  
 compute\_partial\_charges() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper* method), 403  
 compute\_partial\_charges() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper* method), 394  
 compute\_partial\_charges\_am1bcc() (*openforcefield.topology.FrozenMolecule* method), 47  
 compute\_partial\_charges\_am1bcc() (*openforcefield.topology.Molecule* method), 65  
 compute\_partial\_charges\_am1bcc() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper* method), 403  
 compute\_partial\_charges\_am1bcc() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper* method), 394  
 compute\_wiberg\_bond\_orders() (*openforcefield.topology.FrozenMolecule* method), 48  
 compute\_wiberg\_bond\_orders() (*openforcefield.topology.Molecule* method), 65  
 compute\_wiberg\_bond\_orders() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper* method), 395  
 conformers() (*openforcefield.topology.FrozenMolecule* property), 49  
 conformers() (*openforcefield.topology.Molecule* property), 65  
 constrained\_atom\_pairs() (*openforcefield.topology.Topology* property), 86  
 convert\_0\_1\_smirnoff\_to\_0\_2() (in module *openforcefield.utils.utils*), 411  
 convert\_0\_2\_smirnoff\_to\_0\_3() (in module *openforcefield.utils.utils*), 411  
 converter() (*openforcefield.typing.engines.smirnoff.parameters.IndexedParameterAttribute* method), 382  
 converter() (*openforcefield.typing.engines.smirnoff.parameters.ParameterAttribute* method), 380  
 copy() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList* method), 221  
 count() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList* method), 221  
 create\_openmm\_system() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField* method), 153  
 create\_parmed\_structure() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField* method), 153

## D

date() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField* property), 150  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler* method), 260  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler.AngleType* method), 176, 259  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler* method), 244  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler.BondType* method), 167, 243  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler* method), 344  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler* method), 374  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler.GBSAType* method), 218, 373  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler* method), 297  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler.ImproperTorsionType* method), 197, 295  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler* method), 226  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ParameterType* method), 157  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler* method), 278  
 delete\_cosmetic\_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler.ProperTorsionType* method), 187, 277

`delete_cosmetic_attribute()` (openforce-  
*field.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler* (class in openforce-  
*method*), 350 *field.typing.engines.smirnoff.forcefield*),  
`delete_cosmetic_attribute()` (openforce-  
*field.typing.engines.smirnoff.parameters.vdWHandler* *full\_charge()* (openforcefield.topology.Atom prop-  
*method*), 328 *erty*), 107  
`delete_cosmetic_attribute()` (openforce- *fractional\_bond\_order\_model()* (openforce-  
*field.typing.engines.smirnoff.parameters.vdWHandler.vdWHandler* *Topology property*), 86  
*method*), 208, 326 *from\_bson()* (openforcefield.topology.Atom class  
*method*), 108  
*from\_bson()* (openforcefield.topology.Bond class  
*method*), 112  
*from\_bson()* (openforcefield.topology.FrozenMolecule  
class method), 55  
*from\_bson()* (openforcefield.topology.Molecule class  
method), 66  
*from\_bson()* (openforcefield.topology.Particle class  
method), 101  
*from\_bson()* (openforcefield.topology.Topology class  
method), 91  
*from\_bson()* (openforcefield.topology.TopologyAtom  
class method), 123  
*from\_bson()* (openforcefield.topology.TopologyBond  
class method), 129  
*from\_bson()* (openforce-  
*field.topology.TopologyVirtualSite* class  
method), 135  
*from\_bson()* (openforcefield.topology.VirtualSite class  
method), 118  
*from\_bson()* (openforce-  
*field.typing.engines.smirnoff.parameters.ElectrostaticsHandler* (openforce-  
*method*), 344 *field.utils.serialization.Serializable* class  
method), 407  
*from\_dict()* (openforcefield.topology.Atom class  
method), 107  
*from\_dict()* (openforcefield.topology.Bond class  
method), 112  
*from\_dict()* (openforcefield.topology.FrozenMolecule  
class method), 45  
*from\_dict()* (openforcefield.topology.Molecule class  
method), 66  
*from\_dict()* (openforcefield.topology.Particle class  
method), 101  
*from\_dict()* (openforcefield.topology.Topology class  
method), 88  
*from\_dict()* (openforcefield.topology.TopologyAtom  
class method), 123  
*from\_dict()* (openforcefield.topology.TopologyBond  
class method), 129  
*from\_dict()* (openforce-  
*field.topology.TopologyMolecule* class  
method), 99  
*from\_dict()* (openforce-  
*field.topology.TopologyVirtualSite* class  
method), 134

**E**

*ElectrostaticsHandler* (class in openforce-  
*field.typing.engines.smirnoff.parameters*),  
329  
*element()* (openforcefield.topology.Atom property),  
107  
*epsilon()* (openforcefield.topology.VirtualSite prop-  
erty), 118  
*extend()* (openforcefield.typing.engines.smirnoff.parameters.  
*method*), 220

**F**

*find\_matches()* (openforce-  
*field.typing.engines.smirnoff.parameters.AngleHandler* class method), 129  
*method*), 261  
*find\_matches()* (openforce-  
*field.typing.engines.smirnoff.parameters.BondHandler* class method), 135  
*method*), 245  
*find\_matches()* (openforce-  
*field.typing.engines.smirnoff.parameters.ElectrostaticsHandler* (openforce-  
*method*), 344 *field.utils.serialization.Serializable* class  
method), 407  
*find\_matches()* (openforce-  
*field.typing.engines.smirnoff.parameters.GBSAHandler* *from\_dict()* (openforcefield.topology.Atom class  
method), 107  
method), 374  
*find\_matches()* (openforce-  
*field.typing.engines.smirnoff.parameters.ImproperTorsionHandler* *from\_dict()* (openforcefield.topology.Bond class  
method), 112  
method), 296  
*find\_matches()* (openforce-  
*field.typing.engines.smirnoff.parameters.ParameterHandler* *from\_dict()* (openforcefield.topology.FrozenMolecule  
class method), 45  
method), 225  
*find\_matches()* (openforce-  
*field.typing.engines.smirnoff.parameters.ProperTorsionHandler* *from\_dict()* (openforcefield.topology.Molecule class  
method), 101  
method), 279  
*find\_matches()* (openforce-  
*field.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler* (openforcefield.topology.TopologyAtom  
class method), 123  
method), 350  
*find\_matches()* (openforce-  
*field.typing.engines.smirnoff.parameters.vdWHandler* class method), 129  
method), 328  
*find\_smarts\_matches()* (openforce-  
*field.utils.toolkits.OpenEyeToolkitWrapper* class method), 99  
method), 395  
*find\_smarts\_matches()* (openforce-  
*field.utils.toolkits.RDKitToolkitWrapper* class method), 134

`from_dict()` (*openforcefield.topology.VirtualSite* class method), 117  
`from_file()` (*openforcefield.topology.FrozenMolecule* static method), 52  
`from_file()` (*openforcefield.topology.Molecule* static method), 66  
`from_file()` (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper* method), 403  
`from_file()` (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper* method), 391  
`from_file()` (*openforcefield.utils.toolkits.RDKitToolkitWrapper* method), 398  
`from_file()` (*openforcefield.utils.toolkits.ToolkitWrapper* method), 388  
`from_file_obj()` (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper* method), 403  
`from_file_obj()` (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper* method), 392  
`from_file_obj()` (*openforcefield.utils.toolkits.RDKitToolkitWrapper* method), 398  
`from_file_obj()` (*openforcefield.utils.toolkits.ToolkitWrapper* method), 388  
`from_iupac()` (*openforcefield.topology.FrozenMolecule* class method), 50  
`from_iupac()` (*openforcefield.topology.Molecule* class method), 66  
`from_json()` (*openforcefield.topology.Atom* class method), 108  
`from_json()` (*openforcefield.topology.Bond* class method), 113  
`from_json()` (*openforcefield.topology.FrozenMolecule* class method), 55  
`from_json()` (*openforcefield.topology.Molecule* class method), 67  
`from_json()` (*openforcefield.topology.Particle* class method), 101  
`from_json()` (*openforcefield.topology.Topology* class method), 91  
`from_json()` (*openforcefield.topology.TopologyAtom* class method), 124  
`from_json()` (*openforcefield.topology.TopologyBond* class method), 129  
`from_json()` (*openforcefield.topology.TopologyVirtualSite* class method), 135  
`from_json()` (*openforcefield.topology.VirtualSite* class method), 118  
`from_json()` (*openforcefield.utils.serialization.Serializable* class method), 407  
`from_mdtraj()` (*openforcefield.topology.Topology* static method), 89  
`from_messagepack()` (*openforcefield.topology.Atom* class method), 108  
`from_messagepack()` (*openforcefield.topology.Bond* class method), 113  
`from_messagepack()` (*openforcefield.topology.FrozenMolecule* class method), 55  
`from_messagepack()` (*openforcefield.topology.Molecule* class method), 67  
`from_messagepack()` (*openforcefield.topology.Particle* class method), 101  
`from_messagepack()` (*openforcefield.topology.Topology* class method), 91  
`from_messagepack()` (*openforcefield.topology.TopologyAtom* class method), 124  
`from_messagepack()` (*openforcefield.topology.TopologyBond* class method), 129  
`from_messagepack()` (*openforcefield.topology.TopologyVirtualSite* class method), 135  
`from_messagepack()` (*openforcefield.topology.VirtualSite* class method), 118  
`from_messagepack()` (*openforcefield.utils.serialization.Serializable* class method), 408  
`from_molecules()` (*openforcefield.topology.Topology* class method), 85  
`from_object()` (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper* method), 391  
`from_object()` (*openforcefield.utils.toolkits.RDKitToolkitWrapper* method), 397  
`from_openeye()` (*openforcefield.topology.FrozenMolecule* static method), 53  
`from_openeye()` (*openforcefield.topology.Molecule* static method), 67  
`from_openeye()` (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper* static method), 392  
`from_openmm()` (*openforcefield.topology.Topology* class method), 117

method), 88

from\_parmed() (openforcefield.topology.Topology static method), 89

from\_pickle() (openforcefield.topology.Atom class method), 108

from\_pickle() (openforcefield.topology.Bond class method), 113

from\_pickle() (openforcefield.topology.FrozenMolecule class method), 55

from\_pickle() (openforcefield.topology.Molecule class method), 68

from\_pickle() (openforcefield.topology.Particle class method), 102

from\_pickle() (openforcefield.topology.Topology class method), 91

from\_pickle() (openforcefield.topology.TopologyAtom class method), 124

from\_pickle() (openforcefield.topology.TopologyBond class method), 129

from\_pickle() (openforcefield.topology.TopologyVirtualSite class method), 135

from\_pickle() (openforcefield.topology.VirtualSite class method), 118

from\_pickle() (openforcefield.utils.serialization.Serializable class method), 409

from\_rdkit() (openforcefield.topology.FrozenMolecule static method), 53

from\_rdkit() (openforcefield.topology.Molecule static method), 68

from\_rdkit() (openforcefield.utils.toolkits.RDKitToolkitWrapper method), 399

from\_smiles() (openforcefield.topology.FrozenMolecule static method), 46

from\_smiles() (openforcefield.topology.Molecule static method), 68

from\_smiles() (openforcefield.utils.toolkits.OpenEyeToolkitWrapper method), 393

from\_smiles() (openforcefield.utils.toolkits.RDKitToolkitWrapper method), 399

from\_toml() (openforcefield.topology.Atom class method), 109

from\_toml() (openforcefield.topology.Bond class method), 113

from\_toml() (openforcefield.topology.FrozenMolecule class method), 55

from\_toml() (openforcefield.topology.Molecule class method), 69

from\_toml() (openforcefield.topology.Particle class method), 102

from\_toml() (openforcefield.topology.Topology class method), 92

from\_toml() (openforcefield.topology.TopologyAtom class method), 124

from\_toml() (openforcefield.topology.TopologyBond class method), 130

from\_toml() (openforcefield.topology.TopologyVirtualSite class method), 135

from\_toml() (openforcefield.topology.VirtualSite class method), 119

from\_toml() (openforcefield.utils.serialization.Serializable class method), 408

from\_topology() (openforcefield.topology.FrozenMolecule static method), 51

from\_topology() (openforcefield.topology.Molecule static method), 69

from\_xml() (openforcefield.topology.Atom class method), 109

from\_xml() (openforcefield.topology.Bond class method), 113

from\_xml() (openforcefield.topology.FrozenMolecule class method), 56

from\_xml() (openforcefield.topology.Molecule class method), 70

from\_xml() (openforcefield.topology.Particle class method), 102

from\_xml() (openforcefield.topology.Topology class method), 92

from\_xml() (openforcefield.topology.TopologyAtom class method), 124

from\_xml() (openforcefield.topology.TopologyBond class method), 130

from\_xml() (openforcefield.topology.TopologyVirtualSite class method), 136

from\_xml() (openforcefield.topology.VirtualSite class method), 119

from\_xml() (openforcefield.utils.serialization.Serializable class method), 409

from\_yaml() (openforcefield.topology.Atom class method), 109

from\_yaml() (openforcefield.topology.Bond class method), 114

from\_yaml() (openforcefield.topology.FrozenMolecule class method), 56

from\_yaml() (openforcefield.topology.Molecule class method), 70



- [from\\_yaml\(\) \(openforcefield.topology.Particle class method\), 102](#)  
[from\\_yaml\(\) \(openforcefield.topology.Topology class method\), 92](#)  
[from\\_yaml\(\) \(openforcefield.topology.TopologyAtom class method\), 125](#)  
[from\\_yaml\(\) \(openforcefield.topology.TopologyBond class method\), 130](#)  
[from\\_yaml\(\) \(openforcefield.topology.TopologyVirtualSite class method\), 136](#)  
[from\\_yaml\(\) \(openforcefield.topology.VirtualSite class method\), 119](#)  
[from\\_yaml\(\) \(openforcefield.utils.serialization.Serializable class method\), 408](#)  
[FrozenMolecule \(class in openforcefield.topology\), 39](#)
- ## G
- [GBSAHandler \(class in openforcefield.typing.engines.smirnoff.parameters\), 351](#)  
[GBSAHandler.GBSAType \(class in openforcefield.typing.engines.smirnoff.parameters\), 364](#)  
[GBSAType \(class in openforcefield.typing.engines.smirnoff.parameters.GBSAHandler\), 209](#)  
[generate\\_conformers\(\) \(openforcefield.topology.FrozenMolecule method\), 47](#)  
[generate\\_conformers\(\) \(openforcefield.topology.Molecule method\), 70](#)  
[generate\\_conformers\(\) \(openforcefield.utils.toolkits.OpenEyeToolkitWrapper method\), 394](#)  
[generate\\_conformers\(\) \(openforcefield.utils.toolkits.RDKitToolkitWrapper method\), 399](#)  
[get\\_bond\\_between\(\) \(openforcefield.topology.FrozenMolecule method\), 57](#)  
[get\\_bond\\_between\(\) \(openforcefield.topology.Molecule method\), 70](#)  
[get\\_bond\\_between\(\) \(openforcefield.topology.Topology method\), 90](#)  
[get\\_data\\_file\\_path\(\) \(in module openforcefield.utils.utils\), 410](#)  
[get\\_fractional\\_bond\\_order\(\) \(openforcefield.topology.Topology method\), 94](#)  
[get\\_fractional\\_bond\\_orders\(\) \(openforcefield.topology.FrozenMolecule method\), 54](#)  
[get\\_fractional\\_bond\\_orders\(\) \(openforcefield.topology.Molecule method\), 70](#)  
[get\\_molecule\\_parameterIDs\(\) \(in module openforcefield.utils.structure\), 411](#)  
[get\\_parameter\(\) \(openforcefield.typing.engines.smirnoff.parameters.AngleHandler method\), 261](#)  
[get\\_parameter\(\) \(openforcefield.typing.engines.smirnoff.parameters.BondHandler method\), 245](#)  
[get\\_parameter\(\) \(openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method\), 344](#)  
[get\\_parameter\(\) \(openforcefield.typing.engines.smirnoff.parameters.GBSAHandler method\), 374](#)  
[get\\_parameter\(\) \(openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler method\), 297](#)  
[get\\_parameter\(\) \(openforcefield.typing.engines.smirnoff.parameters.ParameterHandler method\), 225](#)  
[get\\_parameter\(\) \(openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method\), 279](#)  
[get\\_parameter\(\) \(openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler method\), 350](#)  
[get\\_parameter\(\) \(openforcefield.typing.engines.smirnoff.parameters.vdWHandler method\), 328](#)  
[get\\_parameter\\_handler\(\) \(openforcefield.typing.engines.smirnoff.forcefield.ForceField method\), 151](#)  
[get\\_parameter\\_io\\_handler\(\) \(openforcefield.typing.engines.smirnoff.forcefield.ForceField method\), 151](#)  
[getAlphaAtoms\(\) \(openforcefield.typing.chemistry.ChemicalEnvironment method\), 145](#)  
[getAlphaBonds\(\) \(openforcefield.typing.chemistry.ChemicalEnvironment method\), 145](#)  
[getANDtypes\(\) \(openforcefield.typing.chemistry.ChemicalEnvironment.Atom method\), 141](#)  
[getANDtypes\(\) \(openforcefield.typing.chemistry.ChemicalEnvironment.Bond method\), 142](#)  
[getAtoms\(\) \(openforcefield.typing.chemistry.ChemicalEnvironment method\), 144](#)  
[getBetaAtoms\(\) \(openforcefield.typing.chemistry.ChemicalEnvironment method\), 144](#)

- method), 145
- getBetaBonds() (openforcefield.typing.chemistry.ChemicalEnvironment method), 145
- getBond() (openforcefield.typing.chemistry.ChemicalEnvironment method), 144
- getBondOrder() (openforcefield.typing.chemistry.ChemicalEnvironment method), 145
- getBonds() (openforcefield.typing.chemistry.ChemicalEnvironment method), 144
- GetComponentList() (openforcefield.typing.chemistry.ChemicalEnvironment method), 143
- getIndexedAtoms() (openforcefield.typing.chemistry.ChemicalEnvironment method), 145
- getIndexedBonds() (openforcefield.typing.chemistry.ChemicalEnvironment method), 145
- getNeighbors() (openforcefield.typing.chemistry.ChemicalEnvironment method), 145
- getOrder() (openforcefield.typing.chemistry.ChemicalEnvironment.Bond method), 142
- getORtypes() (openforcefield.typing.chemistry.ChemicalEnvironment.Atom method), 141
- getORtypes() (openforcefield.typing.chemistry.ChemicalEnvironment.Bond method), 142
- getType() (openforcefield.typing.chemistry.ChemicalEnvironment method), 145
- getUnindexedAtoms() (openforcefield.typing.chemistry.ChemicalEnvironment method), 145
- getUnindexedBonds() (openforcefield.typing.chemistry.ChemicalEnvironment method), 145
- getValence() (openforcefield.typing.chemistry.ChemicalEnvironment method), 145
- I**
- impropers() (openforcefield.topology.FrozenMolecule property), 50
- impropers() (openforcefield.topology.Molecule property), 71
- impropers() (openforcefield.topology.Topology property), 88
- impropers() (openforcefield.topology.TopologyMolecule property), 99
- ImproperTorsionHandler (class in openforcefield.typing.engines.smirnoff.parameters), 280
- ImproperTorsionHandler.ImproperTorsionType (class in openforcefield.typing.engines.smirnoff.parameters), 286
- ImproperTorsionType (class in openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler), 187
- index() (openforcefield.typing.engines.smirnoff.parameters.ParameterList method), 220
- IndexedParameterAttribute (class in openforcefield.typing.engines.smirnoff.parameters), 381
- IndexedParameterAttribute.UNDEFINED (class in openforcefield.typing.engines.smirnoff.parameters), 382
- inherit\_docstrings() (in module openforcefield.utils.utils), 410
- insert() (openforcefield.typing.engines.smirnoff.parameters.ParameterList method), 220
- is\_aromatic() (openforcefield.topology.Atom property), 107
- is\_available() (openforcefield.utils.toolkits.AmberToolsToolkitWrapper static method), 402
- is\_available() (openforcefield.utils.toolkits.OpenEyeToolkitWrapper static method), 391
- is\_available() (openforcefield.utils.toolkits.RDKitToolkitWrapper static method), 397
- is\_available() (openforcefield.utils.toolkits.ToolkitWrapper static method), 388
- is\_bonded() (openforcefield.topology.Topology method), 90
- is\_bonded\_to() (openforcefield.topology.Atom method), 107
- is\_constrained() (openforcefield.topology.Topology method), 94
- is\_isomorphic() (openforcefield.topology.FrozenMolecule method), 46
- is\_isomorphic() (openforcefield.topology.Molecule method), 71
- isAlpha() (openforcefield.typing.chemistry.ChemicalEnvironment method), 145

`isBeta()` (`openforcefield.typing.chemistry.ChemicalEnvironment` method), 145  
`isIndexed()` (`openforcefield.typing.chemistry.ChemicalEnvironment` method), 145  
`isUnindexed()` (`openforcefield.typing.chemistry.ChemicalEnvironment` method), 145  
`isValid()` (`openforcefield.typing.chemistry.ChemicalEnvironment` method), 143

## K

`known_kwargs()` (`openforcefield.typing.engines.smirnoff.parameters.AngleHandler` property), 261  
`known_kwargs()` (`openforcefield.typing.engines.smirnoff.parameters.BondHandler` property), 245  
`known_kwargs()` (`openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler` property), 345  
`known_kwargs()` (`openforcefield.typing.engines.smirnoff.parameters.GBSAHandler` property), 374  
`known_kwargs()` (`openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler` property), 297  
`known_kwargs()` (`openforcefield.typing.engines.smirnoff.parameters.ParametersHandler` property), 224  
`known_kwargs()` (`openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler` property), 279  
`known_kwargs()` (`openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler` property), 350  
`known_kwargs()` (`openforcefield.typing.engines.smirnoff.parameters.vdWHandler` property), 328

## L

`label_molecules()` (`openforcefield.typing.engines.smirnoff.forcefield.ForceField` method), 153

## M

`mass()` (`openforcefield.topology.Atom` property), 107  
`Molecule` (class in `openforcefield.topology`), 57  
`molecule()` (`openforcefield.topology.Atom` property), 109  
`molecule()` (`openforcefield.topology.Particle` property), 101  
`molecule()` (`openforcefield.topology.TopologyAtom` property), 123  
`molecule()` (`openforcefield.topology.TopologyBond` property), 128  
`molecule()` (`openforcefield.topology.TopologyVirtualSite` property), 134  
`molecule()` (`openforcefield.topology.VirtualSite` property), 119  
`molecule_atom_index()` (`openforcefield.topology.Atom` property), 108  
`molecule_bond_index()` (`openforcefield.topology.Bond` property), 112  
`molecule_particle_index()` (`openforcefield.topology.Atom` property), 108  
`molecule_particle_index()` (`openforcefield.topology.Particle` property), 101  
`molecule_particle_index()` (`openforcefield.topology.VirtualSite` property), 118  
`molecule_virtual_site_index()` (`openforcefield.topology.VirtualSite` property), 117

## N

`n_angles()` (`openforcefield.topology.FrozenMolecule` property), 49  
`n_angles()` (`openforcefield.topology.Molecule` property), 71  
`n_angles()` (`openforcefield.topology.Topology` property), 87  
`n_angles()` (`openforcefield.topology.TopologyMolecule` property), 98  
`n_atoms()` (`openforcefield.topology.FrozenMolecule` property), 49  
`n_atoms()` (`openforcefield.topology.Molecule` property), 71  
`n_atoms()` (`openforcefield.topology.TopologyMolecule` property), 97  
`n_bonds()` (`openforcefield.topology.FrozenMolecule` property), 49  
`n_bonds()` (`openforcefield.topology.Molecule` property), 71  
`n_bonds()` (`openforcefield.topology.TopologyMolecule` property), 97  
`n_conformers()` (`openforcefield.topology.FrozenMolecule` property), 49  
`n_conformers()` (`openforcefield.topology.Molecule` property), 71  
`n_impropers()` (`openforcefield.topology.FrozenMolecule` property), 49  
`n_impropers()` (`openforcefield.topology.Molecule` property), 71

`n_impropers()` (*openforcefield.topology.Topology* property), 88  
`n_impropers()` (*openforcefield.topology.TopologyMolecule* property), 99  
`n_particles()` (*openforcefield.topology.FrozenMolecule* property), 49  
`n_particles()` (*openforcefield.topology.Molecule* property), 71  
`n_particles()` (*openforcefield.topology.TopologyMolecule* property), 98  
`n_propers()` (*openforcefield.topology.FrozenMolecule* property), 49  
`n_propers()` (*openforcefield.topology.Molecule* property), 71  
`n_propers()` (*openforcefield.topology.Topology* property), 88  
`n_propers()` (*openforcefield.topology.TopologyMolecule* property), 99  
`n_reference_molecules()` (*openforcefield.topology.Topology* property), 86  
`n_topology_atoms()` (*openforcefield.topology.Topology* property), 87  
`n_topology_bonds()` (*openforcefield.topology.Topology* property), 87  
`n_topology_molecules()` (*openforcefield.topology.Topology* property), 86  
`n_topology_particles()` (*openforcefield.topology.Topology* property), 87  
`n_topology_virtual_sites()` (*openforcefield.topology.Topology* property), 87  
`n_virtual_sites()` (*openforcefield.topology.FrozenMolecule* property), 49  
`n_virtual_sites()` (*openforcefield.topology.Molecule* property), 71  
`n_virtual_sites()` (*openforcefield.topology.TopologyMolecule* property), 98  
`name()` (*openforcefield.topology.Atom* property), 107  
`name()` (*openforcefield.topology.FrozenMolecule* property), 50  
`name()` (*openforcefield.topology.Molecule* property), 72  
`name()` (*openforcefield.topology.Particle* property), 101  
`name()` (*openforcefield.topology.VirtualSite* property), 118

## O

`OpenEyeToolkitWrapper` (class in *openforce-*

*field.utils.toolkits*), 389

## P

`ParameterAttribute` (class in *openforcefield.typing.engines.smirnoff.parameters*), 378

`ParameterAttribute.UNDEFINED` (class in *openforcefield.typing.engines.smirnoff.parameters*), 380

`ParameterHandler` (class in *openforcefield.typing.engines.smirnoff.parameters*), 221

`ParameterIOHandler` (class in *openforcefield.typing.engines.smirnoff.io*), 375

`ParameterList` (class in *openforcefield.typing.engines.smirnoff.parameters*), 219

`parameters()` (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler* property), 261

`parameters()` (*openforcefield.typing.engines.smirnoff.parameters.BondHandler* property), 245

`parameters()` (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler* property), 345

`parameters()` (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler* property), 374

`parameters()` (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler* property), 297

`parameters()` (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler* property), 224

`parameters()` (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler* property), 279

`parameters()` (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler* property), 350

`parameters()` (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler* property), 328

`ParameterType` (class in *openforcefield.typing.engines.smirnoff.parameters*), 154

`parse_file()` (*openforcefield.typing.engines.smirnoff.io.ParameterIOHandler* method), 376

`parse_file()` (*openforcefield.typing.engines.smirnoff.io.XMLParameterIOHandler* method), 377

`parse_smirnoff_from_source()` (*openforce-*



*field.typing.engines.smirnoff.forcefield.ForceField*.*postprocess\_system()* (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler* method), 152  
*parse\_sources()* (*openforcefield.typing.engines.smirnoff.forcefield.ForceField*.*postprocess\_system()* method), 151  
*parse\_string()* (*openforcefield.typing.engines.smirnoff.io.ParameterIOHandler*.*properties()* method), 376  
*parse\_string()* (*openforcefield.typing.engines.smirnoff.io.XMLParameterIOHandler*.*properties()* method), 377  
*partial\_charge()* (*openforcefield.topology.Atom* property), 107  
*partial\_charges()* (*openforcefield.topology.FrozenMolecule* property), 49  
*partial\_charges()* (*openforcefield.topology.Molecule* property), 72  
*Particle* (class in *openforcefield.topology*), 99  
*particle()* (*openforcefield.topology.TopologyMolecule* method), 98  
*particle\_start\_topology\_index()* (*openforcefield.topology.TopologyMolecule* property), 98  
*particles()* (*openforcefield.topology.FrozenMolecule* property), 49  
*particles()* (*openforcefield.topology.Molecule* property), 72  
*particles()* (*openforcefield.topology.TopologyMolecule* property), 98  
*pop()* (*openforcefield.typing.engines.smirnoff.parameters.ParameterListWrapper* (class in *openforcefield.utils.toolkits*), 396 method), 221  
*postprocess\_system()* (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler* method), 261  
*postprocess\_system()* (*openforcefield.typing.engines.smirnoff.parameters.BondHandler* method), 245  
*postprocess\_system()* (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler* method), 345  
*postprocess\_system()* (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler* method), 374  
*postprocess\_system()* (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler* method), 297  
*postprocess\_system()* (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler* method), 225  
*postprocess\_system()* (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler* method), 279  
*postprocess\_system()* (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler* method), 349  
*postprocess\_system()* (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler* method), 327  
*properties()* (*openforcefield.topology.FrozenMolecule* property), 49  
*properties()* (*openforcefield.topology.Molecule* property), 72  
*properties()* (*openforcefield.topology.Topology* property), 88  
*properties()* (*openforcefield.topology.TopologyMolecule* property), 98  
*properties()* (*openforcefield.topology.FrozenMolecule* property), 50  
*properties()* (*openforcefield.topology.Molecule* property), 72  
*ProperTorsionHandler* (class in *openforcefield.typing.engines.smirnoff.parameters*), 262  
*ProperTorsionHandler.ProperTorsionType* (class in *openforcefield.typing.engines.smirnoff.parameters*), 268  
*ProperTorsionType* (class in *openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler*), 177

## R

*ParameterListWrapper* (class in *openforcefield.utils.toolkits*), 396  
*reference\_molecule()* (*openforcefield.topology.TopologyMolecule* property), 97  
*reference\_molecules()* (*openforcefield.topology.Topology* property), 85  
*register\_parameter\_handler()* (*openforcefield.typing.engines.smirnoff.forcefield.ForceField* method), 150  
*register\_parameter\_io\_handler()* (*openforcefield.typing.engines.smirnoff.forcefield.ForceField* method), 150  
*register\_toolkit()* (*openforcefield.utils.toolkits.ToolkitRegistry* method), 155  
*registered\_toolkits()* (*openforcefield.utils.toolkits.ToolkitRegistry* property), 155  
*remove()* (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler* method), 221  
*remove\_from()* (*openforcefield.typing.chemistry.ChemicalEnvironment* method), 279

method), 144  
 resolve() (openforcefield.utils.toolkits.ToolkitRegistry  
 method), 385  
 reverse() (openforce-  
 field.typing.engines.smirnoff.parameters.Parameters  
 method), 221  
 rmin\_half() (openforcefield.topology.VirtualSite  
 property), 118

## S

selectAtom() (openforce-  
 field.typing.chemistry.ChemicalEnvironment  
 method), 143  
 selectBond() (openforce-  
 field.typing.chemistry.ChemicalEnvironment  
 method), 143  
 Serializable (class in openforce-  
 field.utils.serialization), 404  
 setANDtypes() (openforce-  
 field.typing.chemistry.ChemicalEnvironment.Atom  
 method), 141  
 setANDtypes() (openforce-  
 field.typing.chemistry.ChemicalEnvironment.Bond  
 method), 142  
 setORtypes() (openforce-  
 field.typing.chemistry.ChemicalEnvironment.Atom  
 method), 141  
 setORtypes() (openforce-  
 field.typing.chemistry.ChemicalEnvironment.Bond  
 method), 142  
 sigma() (openforcefield.topology.VirtualSite prop-  
 erty), 118  
 sort() (openforcefield.typing.engines.smirnoff.parameters.Parameters  
 method), 221  
 stereochemistry() (openforcefield.topology.Atom  
 property), 107

## T

temporary\_cd() (in module openforcefield.utils.utils),  
 410  
 temporary\_directory() (in module openforce-  
 field.utils.utils), 410  
 to\_bson() (openforcefield.topology.Atom method),  
 109  
 to\_bson() (openforcefield.topology.Bond method),  
 114  
 to\_bson() (openforcefield.topology.FrozenMolecule  
 method), 56  
 to\_bson() (openforcefield.topology.Molecule method),  
 72  
 to\_bson() (openforcefield.topology.Particle method),  
 102  
 to\_bson() (openforcefield.topology.Topology method),  
 92

to\_bson() (openforcefield.topology.TopologyAtom  
 method), 125

to\_bson() (openforcefield.topology.TopologyBond  
 method), 130

to\_bson() (openforce-  
 field.topology.TopologyVirtualSite method),  
 136

to\_bson() (openforcefield.topology.VirtualSite  
 method), 119

to\_bson() (openforce-  
 field.utils.serialization.Serializable method),  
 407

to\_dict() (openforcefield.topology.Atom method),  
 107

to\_dict() (openforcefield.topology.Bond method),  
 112

to\_dict() (openforcefield.topology.FrozenMolecule  
 method), 45

to\_dict() (openforcefield.topology.Molecule method),  
 72

to\_dict() (openforcefield.topology.Particle method),  
 101

to\_dict() (openforcefield.topology.Topology method),  
 88

to\_dict() (openforcefield.topology.TopologyAtom  
 method), 123

to\_dict() (openforcefield.topology.TopologyBond  
 method), 129

to\_dict() (openforcefield.topology.TopologyMolecule  
 method), 99

to\_dict() (openforce-  
 field.topology.TopologyVirtualSite method),  
 131

to\_dict() (openforcefield.topology.VirtualSite  
 method), 117

to\_dict() (openforce-  
 field.typing.engines.smirnoff.parameters.AngleHandler  
 method), 261

to\_dict() (openforce-  
 field.typing.engines.smirnoff.parameters.AngleHandler.AngleType  
 method), 177, 259

to\_dict() (openforce-  
 field.typing.engines.smirnoff.parameters.BondHandler  
 method), 245

to\_dict() (openforce-  
 field.typing.engines.smirnoff.parameters.BondHandler.BondType  
 method), 167, 243

to\_dict() (openforce-  
 field.typing.engines.smirnoff.parameters.ElectrostaticsHandler  
 method), 345

to\_dict() (openforce-  
 field.typing.engines.smirnoff.parameters.GBSAHandler  
 method), 374

to\_dict() (openforce-

Index	427
-------	-----

*field.topology.VirtualSite* method), 120  
 to\_messagepack() (*openforcefield.utils.serialization.Serializable* method), 408  
 to\_networkx() (*openforcefield.topology.FrozenMolecule* method), 48  
 to\_networkx() (*openforcefield.topology.Molecule* method), 73  
 to\_openeye() (*openforcefield.topology.FrozenMolecule* method), 54  
 to\_openeye() (*openforcefield.topology.Molecule* method), 74  
 to\_openeye() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper* static method), 393  
 to\_openmm() (*openforcefield.topology.Topology* method), 89  
 to\_parmed() (*openforcefield.topology.Topology* method), 90  
 to\_pickle() (*openforcefield.topology.Atom* method), 110  
 to\_pickle() (*openforcefield.topology.Bond* method), 114  
 to\_pickle() (*openforcefield.topology.FrozenMolecule* method), 57  
 to\_pickle() (*openforcefield.topology.Molecule* method), 74  
 to\_pickle() (*openforcefield.topology.Particle* method), 103  
 to\_pickle() (*openforcefield.topology.Topology* method), 93  
 to\_pickle() (*openforcefield.topology.TopologyAtom* method), 125  
 to\_pickle() (*openforcefield.topology.TopologyBond* method), 131  
 to\_pickle() (*openforcefield.topology.TopologyVirtualSite* method), 136  
 to\_pickle() (*openforcefield.topology.VirtualSite* method), 120  
 to\_pickle() (*openforcefield.utils.serialization.Serializable* method), 409  
 to\_rdkit() (*openforcefield.topology.FrozenMolecule* method), 53  
 to\_rdkit() (*openforcefield.topology.Molecule* method), 74  
 to\_rdkit() (*openforcefield.utils.toolkits.RDKitToolkitWrapper* class method), 400  
 to\_smiles() (*openforcefield.topology.FrozenMolecule* method), 45  
 to\_smiles() (*openforcefield.topology.Molecule* method), 75  
 to\_smiles() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper* static method), 393  
 to\_smiles() (*openforcefield.utils.toolkits.RDKitToolkitWrapper* class method), 399  
 to\_string() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField* method), 152  
 to\_string() (*openforcefield.typing.engines.smirnoff.io.ParameterIOHandler* method), 376  
 to\_string() (*openforcefield.typing.engines.smirnoff.io.XMLParameterIOHandler* method), 377  
 to\_toml() (*openforcefield.topology.Atom* method), 110  
 to\_toml() (*openforcefield.topology.Bond* method), 114  
 to\_toml() (*openforcefield.topology.FrozenMolecule* method), 57  
 to\_toml() (*openforcefield.topology.Molecule* method), 75  
 to\_toml() (*openforcefield.topology.Particle* method), 103  
 to\_toml() (*openforcefield.topology.Topology* method), 93  
 to\_toml() (*openforcefield.topology.TopologyAtom* method), 126  
 to\_toml() (*openforcefield.topology.TopologyBond* method), 131  
 to\_toml() (*openforcefield.topology.TopologyVirtualSite* method), 137  
 to\_toml() (*openforcefield.topology.VirtualSite* method), 120  
 to\_toml() (*openforcefield.utils.serialization.Serializable* method), 408  
 to\_topology() (*openforcefield.topology.FrozenMolecule* method), 51  
 to\_topology() (*openforcefield.topology.Molecule* method), 75  
 to\_xml() (*openforcefield.topology.Atom* method), 110  
 to\_xml() (*openforcefield.topology.Bond* method), 115  
 to\_xml() (*openforcefield.topology.FrozenMolecule* method), 57  
 to\_xml() (*openforcefield.topology.Molecule* method), 75  
 to\_xml() (*openforcefield.topology.Particle* method), 103



- to\_xml() (*openforcefield.topology.Topology* method), 93
- to\_xml() (*openforcefield.topology.TopologyAtom* method), 126
- to\_xml() (*openforcefield.topology.TopologyBond* method), 131
- to\_xml() (*openforcefield.topology.TopologyVirtualSite* method), 137
- to\_xml() (*openforcefield.topology.VirtualSite* method), 120
- to\_xml() (*openforcefield.utils.serialization.Serializable* method), 409
- to\_yaml() (*openforcefield.topology.Atom* method), 110
- to\_yaml() (*openforcefield.topology.Bond* method), 115
- to\_yaml() (*openforcefield.topology.FrozenMolecule* method), 57
- to\_yaml() (*openforcefield.topology.Molecule* method), 76
- to\_yaml() (*openforcefield.topology.Particle* method), 103
- to\_yaml() (*openforcefield.topology.Topology* method), 93
- to\_yaml() (*openforcefield.topology.TopologyAtom* method), 126
- to\_yaml() (*openforcefield.topology.TopologyBond* method), 131
- to\_yaml() (*openforcefield.topology.TopologyVirtualSite* method), 137
- to\_yaml() (*openforcefield.topology.VirtualSite* method), 120
- to\_yaml() (*openforcefield.utils.serialization.Serializable* method), 408
- toolkit\_file\_read\_formats() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper* property), 404
- toolkit\_file\_read\_formats() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper* property), 395
- toolkit\_file\_read\_formats() (*openforcefield.utils.toolkits.RDKitToolkitWrapper* property), 401
- toolkit\_file\_read\_formats() (*openforcefield.utils.toolkits.ToolkitWrapper* property), 388
- toolkit\_file\_write\_formats() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper* property), 404
- toolkit\_file\_write\_formats() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper* property), 395
- toolkit\_file\_write\_formats() (*openforcefield.utils.toolkits.RDKitToolkitWrapper* property), 401
- toolkit\_file\_write\_formats() (*openforcefield.utils.toolkits.ToolkitWrapper* property), 388
- toolkit\_file\_write\_formats() (*openforcefield.utils.toolkits.ToolkitWrapper* property), 401
- toolkit\_file\_write\_formats() (*openforcefield.utils.toolkits.ToolkitWrapper* property), 387
- ToolkitAM1BCCHandler (class in *openforcefield.typing.engines.smirnoff.parameters*), 345
- ToolkitRegistry (class in *openforcefield.utils.toolkits*), 383
- ToolkitWrapper (class in *openforcefield.utils.toolkits*), 386
- Topology (class in *openforcefield.topology*), 79
- topology() (*openforcefield.topology.TopologyMolecule* property), 97
- topology\_atom\_index() (*openforcefield.topology.TopologyAtom* property), 123
- topology\_atoms() (*openforcefield.topology.Topology* property), 87
- topology\_bond\_index() (*openforcefield.topology.TopologyBond* property), 128
- topology\_bonds() (*openforcefield.topology.Topology* property), 87
- topology\_bonds() (*openforcefield.topology.TopologyAtom* property), 123
- topology\_molecule() (*openforcefield.topology.TopologyMolecule* property), 97

[field.topology.TopologyAtom](#) (property), [vdWType](#) (class in [openforcefield.typing.engines.smirnoff.parameters.vdWHandler](#)),  
[123](#) [197](#)  
[topology\\_molecule\(\)](#) ([openforcefield.topology.TopologyBond](#) property), [virtual\\_site\(\)](#) ([openforcefield.topology.Topology](#)  
[128](#) [method](#)), [90](#)  
[topology\\_molecule\(\)](#) ([openforcefield.topology.TopologyVirtualSite](#) property), [virtual\\_site\(\)](#) ([openforcefield.topology.TopologyMolecule](#)  
[134](#) [method](#)), [98](#)  
[topology\\_molecules\(\)](#) ([openforcefield.topology.Topology](#) property), [86](#) [virtual\\_site\(\)](#) ([openforcefield.topology.TopologyVirtualSite](#) property),  
[topology\\_particle\\_index\(\)](#) ([openforcefield.topology.TopologyAtom](#) property), [134](#)  
[123](#) [virtual\\_site\\_start\\_topology\\_index\(\)](#) ([openforcefield.topology.TopologyMolecule](#) property),  
[topology\\_particle\\_index\(\)](#) ([openforcefield.topology.TopologyVirtualSite](#) property), [99](#)  
[134](#) [virtual\\_sites\(\)](#) ([openforcefield.topology.Atom](#) property), [108](#)  
[topology\\_particles\(\)](#) ([openforcefield.topology.Topology](#) property), [87](#) [virtual\\_sites\(\)](#) ([openforcefield.topology.FrozenMolecule](#) property),  
[topology\\_virtual\\_site\\_index\(\)](#) ([openforcefield.topology.TopologyVirtualSite](#) property), [49](#)  
[134](#) [virtual\\_sites\(\)](#) ([openforcefield.topology.Molecule](#) property), [76](#)  
[topology\\_virtual\\_sites\(\)](#) ([openforcefield.topology.Topology](#) property), [87](#) [virtual\\_sites\(\)](#) ([openforcefield.topology.TopologyMolecule](#) property),  
[TopologyAtom](#) (class in [openforcefield.topology](#)), [121](#) [98](#)  
[TopologyBond](#) (class in [openforcefield.topology](#)), [126](#) [VirtualSite](#) (class in [openforcefield.topology](#)), [115](#)  
[TopologyMolecule](#) (class in [openforcefield.topology](#)), [94](#)  
[TopologyVirtualSite](#) (class in [openforcefield.topology](#)), [131](#)  
[torsions\(\)](#) ([openforcefield.topology.FrozenMolecule](#) property), [49](#)  
[torsions\(\)](#) ([openforcefield.topology.Molecule](#) property), [76](#)  
[total\\_charge\(\)](#) ([openforcefield.topology.FrozenMolecule](#) property), [50](#)  
[total\\_charge\(\)](#) ([openforcefield.topology.Molecule](#) property), [76](#)  
[type\(\)](#) ([openforcefield.topology.TopologyVirtualSite](#) property), [134](#)  
[type\(\)](#) ([openforcefield.topology.VirtualSite](#) property), [118](#)

## V

[validate\(\)](#) ([openforcefield.typing.chemistry.ChemicalEnvironment](#) static method), [143](#)  
[vdWHandler](#) (class in [openforcefield.typing.engines.smirnoff.parameters](#)), [298](#)  
[vdWHandler.vdWType](#) (class in [openforcefield.typing.engines.smirnoff.parameters](#)), [316](#)

## X

[XMLParameterIOHandler](#) (class in [openforcefield.typing.engines.smirnoff.io](#)), [376](#)