

---

# OpenFF Units

The Open Force Field Initiative

Apr 24, 2024



**CONTENTS**

<b>1</b>	<b>Installation</b>	<b>3</b>
<b>2</b>	<b>Using OpenFF Units</b>	<b>5</b>
<b>3</b>	<b>Current development</b>	<b>9</b>
3.1	Behavior changes . . . . .	9
	<b>Python Module Index</b>	<b>25</b>
	<b>Index</b>	<b>27</b>



Units of measure for biomolecular software.

OpenFF Units is based on [Pint](#). Its *Quantity*, *Unit*, and *Measurement* types inherit from Pint's, and add improved support for serialization and deserialization. OpenFF Units improves support for biomolecular software by providing a *system of units* that are compatible with [OpenMM](#) and *providing functions* to convert to OpenMM units and back. It also provides *atomic masses* with units, as well as some *other useful maps*.



## INSTALLATION

We recommend installing OpenFF Units with the [Conda](#) or [Mamba](#) package managers. If you don't yet have a Conda distribution installed, we recommend [MambaForge](#) for most users; see the [OpenFF install docs](#). The `openff-units` package can be installed from Conda Forge:

```
conda install -c conda-forge openff-units
```





## USING OPENFF UNITS

OpenFF Units provides the *Quantity* class, which represents a numerical value with units. A *Quantity* can be created by providing a value and units:

```
>>> from openff.units import unit, Quantity
>>>
>>> Quantity(1.007, unit.amu)
<Quantity(1.007, 'unified_atomic_mass_unit')>
```

The *unit* singleton value is a registry of units, but also exposes the *Quantity*, *Unit*, and *Measurement* classes so you don't have to import them individually. Even easier, multiplying a number by the appropriate unit also provides a *Quantity*:

```
>>> mass_proton = 1.007 * unit.amu
>>> mass_proton == unit.Quantity(1.007, unit.amu)
True
```

*Quantity* can also wrap NumPy arrays. It's best to wrap an array of floats in a quantity, rather than have an array of quantities:

```
>>> import numpy as np
>>>
>>> box_vectors = np.array([
...     [5.0, 0.0, 0.0],
...     [0.0, 5.0, 0.0],
...     [0.0, 0.0, 5.0],
... ]) * unit.nanometer
```

When constructed like this, *Quantity* is transparent; it will pass any attributes it doesn't have through to the inner value. This means that a quantity-wrapped array can be used exactly as though it were an array — the units are just checked silently in the background:

```
>>> from numpy.random import rand
>>>
>>> trajectory = 10 * rand(10, 10000, 3) * unit.nanometer
>>> centroids = trajectory.mean(axis=1)[..., None]
>>> last_water = trajectory[:, 97:99, :]
>>> last_water_recentered = last_water - centroids
```

This transparency works with most container types, so it's usually best to have *Quantity* be the outermost wrapper type.

Complex units can be constructed by combining units with the usual arithmetic operations:

```
>>> boltzmann_constant = 8.314462618e-3 * unit.kilojoule / unit.kelvin / unit.avogadro_
↳number
```

Some common constants are provided as units as well:

```
>>> boltzmann_constant = 1.0 * unit.boltzmann_constant
```

Adding or subtracting different units with the same dimensions just works:

```
>>> 1.0 * unit.angstrom + 1.0 * unit.nanometer
<Quantity(11.0, 'angstrom')>
```

But quantities with different dimensions raise an exception:

```
>>> 1.0 * unit.angstrom + 1.0 * unit.nanojoule
Traceback (most recent call last):
...
pint.errors.DimensionalityError: Cannot convert from 'angstrom' ([length]) to 'nanojoule
↳' ([length] ** 2 * [mass] / [time] ** 2)
```

Quantities can be converted between units with the `.to()` method:

```
>>> (1.0 * unit.nanometer).to(unit.angstrom)
<Quantity(10.0, 'angstrom')>
```

Or with the `.ito()` method for in-place transformations:

```
>>> quantity = 10.0 * unit.angstrom
>>> quantity.ito(unit.nanometer)
>>> quantity
<Quantity(1.0, 'nanometer')>
```

The underlying value without units can be retrieved with the `.m` or `.magnitude` properties. Just make sure it's in the units you expect first:

```
>>> quantity = (1.0 * unit.k_B).to_base_units()
>>> assert quantity.units == unit.kilogram * unit.meter**2 / unit.kelvin / unit.second**2
>>> quantity.magnitude
1.380649e-23
```

Alternatively, specify the target units of the output magnitude with `.m_as`:

```
>>> quantity = 1.0 * unit.k_B
>>> quantity.m_as(unit.kilogram * unit.meter**2 / unit.kelvin / unit.second**2)
1.380649e-23
```

OpenFF Units also provides the `from_openmm` and `to_openmm` functions to convert between OpenFF quantities and OpenMM quantities:

```
>>> from openff.units.openmm import from_openmm, to_openmm
>>>
>>> quantity = 10.0 * unit.angstrom
>>> omm_quant = to_openmm(quantity)
>>> omm_quant
```

(continues on next page)

(continued from previous page)

```
Quantity(value=10.0, unit=angstrom)
>>> type(omm_quant)
<class 'openmm.unit.quantity.Quantity'>
>>> quant_roundtrip = from_openmm(omm_quant)
>>> quant_roundtrip
<Quantity(10.0, 'angstrom')>
>>> type(quant_roundtrip)
<class 'openff.units.units.Quantity'>
```

For more details, see the [API reference](#).



## CURRENT DEVELOPMENT

### 3.1 Behavior changes

- #62 Drops support for Python 3.8, following [NEP 29](#).

---

*openff.units*

---

#### 3.1.1 openff.units

##### Module Attributes

---

<i>unit</i>	Registry of units provided by OpenFF Units.
-------------	---

---

##### `unit`

`openff.units.unit: openff.units.units.UnitRegistry`

Registry of units provided by OpenFF Units.

`unit` may be used similarly to a module. It makes constants and units of measure available as attributes. Available units can be found in the `constants` and `defaults` data files.

##### Functions

---

*Measurement*

---

---

<i>ensure_quantity</i>	Given a quantity that could be of a variety of types, attempt to coerce into a given type.
------------------------	--

---

### Measurement

`openff.units.Measurement(*args, **kwargs)`

### ensure\_quantity

`openff.units.ensure_quantity(unknown_quantity: Quantity, type_to_ensure: Literal['openmm', 'openff']) → Quantity`

Given a quantity that could be of a variety of types, attempt to coerce into a given type.

### Examples

```
>>> import numpy
>>> from openmm import unit as openmm_unit
>>> from openff.units import unit
>>> from openff.units.openmm import ensure_quantity
>>> # Create a 9 Angstrom quantity with each registry
>>> length1 = unit.Quantity(9.0, unit.angstrom)
>>> length2 = openmm_unit.Quantity(9.0, openmm_unit.angstrom)
>>> # Similar quantities are be coerced into requested type
>>> assert type(ensure_quantity(length1, "openmm")) == openmm_unit.Quantity
>>> assert type(ensure_quantity(length2, "openff")) == unit.Quantity
>>> # Seemingly-redundant "conversions" short-circuit
>>> assert ensure_quantity(length1, "openff") == ensure_quantity(length2, "openff")
>>> assert ensure_quantity(length1, "openmm") == ensure_quantity(length2, "openmm")
>>> # NumPy arrays and some primitives are automatically up-converted to `Quantity`
↳objects
>>> # Note that their units are set to "dimensionless"
>>> ensure_quantity(numpy.array([1, 2]), "openff")
<Quantity([1 2], 'dimensionless')>
>>> ensure_quantity(4.0, "openmm")
Quantity(value=4.0, unit=dimensionless)
```

### Classes

---

*Quantity*

---

*Unit*

---

## Quantity

```
class openff.units.Quantity(value: MagnitudeT, units: Optional[UnitLike] = None)
```

```
class openff.units.Quantity(value: str, units: Optional[UnitLike] = None)
```

```
class openff.units.Quantity(value: Sequence[ScalarT], units: Optional[UnitLike] = None)
```

```
class openff.units.Quantity(value: PlainQuantity[Any], units: Optional[UnitLike] = None)
```

Bases: `Quantity`

### Methods

<code>check</code>	Return true if the quantity's dimension matches passed dimension.
<code>clip</code>	
<code>compare</code>	
<code>compatible_units</code>	
<code>compute</code>	Compute the Dask array wrapped by pint.PlainQuantity.
<code>dot</code>	Dot product of two arrays.
<code>fill</code>	
<code>format_babel</code>	
<code>from_list</code>	Transforms a list of Quantities into an numpy.array quantity.
<code>from_sequence</code>	Transforms a sequence of Quantities into an numpy.array quantity.
<code>from_tuple</code>	
<code>is_compatible_with</code>	check if the other object is compatible
<code>ito</code>	Inplace rescale to different units.
<code>ito_base_units</code>	Return PlainQuantity rescaled to plain units.
<code>ito_preferred</code>	Return Quantity converted to a unit composed of the preferred units.
<code>ito_reduced_units</code>	Return PlainQuantity scaled in place to reduced units, i.e. one unit per dimension.
<code>ito_root_units</code>	Return PlainQuantity rescaled to root units.
<code>m_as</code>	PlainQuantity's magnitude expressed in particular units.
<code>persist</code>	Persist the Dask Array wrapped by pint.PlainQuantity.
<code>plus_minus</code>	
<code>prod</code>	Return the product of quantity elements over a given axis
<code>put</code>	

continues on next page

Table 1 – continued from previous page

<i>searchsorted</i>	
<i>to</i>	Return PlainQuantity rescaled to different units.
<i>to_base_units</i>	Return PlainQuantity rescaled to plain units.
<i>to_compact</i>	"Return PlainQuantity rescaled to compact, human-readable units.
<i>to_openmm</i>	Convert the quantity to an openmm.unit.Quantity.
<i>to_preferred</i>	Return Quantity converted to a unit composed of the preferred units.
<i>to_reduced_units</i>	Return PlainQuantity scaled in place to reduced units, i.e. one unit per dimension.
<i>to_root_units</i>	Return PlainQuantity rescaled to root units.
<i>to_timedelta</i>	
<i>to_tuple</i>	
<i>tolist</i>	
<i>visualize</i>	Produce a visual representation of the Dask graph.



**Attributes**

<i>T</i>	
<i>UnitsContainer</i>	
<i>default_format</i>	Default formatting string.
<i>dimensionality</i>	returns: <code>dict</code> -- Dimensionality of the PlainQuantity, e.g.
<i>dimensionless</i>	
<i>flat</i>	
<i>force_ndarray</i>	
<i>force_ndarray_like</i>	
<i>imag</i>	
<i>m</i>	PlainQuantity's magnitude.
<i>magnitude</i>	PlainQuantity's magnitude.
<i>ndim</i>	
<i>real</i>	
<i>shape</i>	
<i>u</i>	PlainQuantity's units.
<i>unitless</i>	
<i>units</i>	PlainQuantity's units.

**property T****property UnitsContainer:** `Callable[..., UnitsContainerT]`**check**(*dimension: UnitLike*) → `bool`

Return true if the quantity's dimension matches passed dimension.

**clip**(*min=None, max=None, out=None, \*\*kwargs*)**compare**(*\*args, \*\*kwargs*)**compatible\_units**(*\*contexts*)**compute**(*\*\*kwargs*)

Compute the Dask array wrapped by pint.PlainQuantity.

**Parameters** *\*\*kwargs* (`dict`) – Any keyword arguments to pass to `dask.compute`.**Returns** `pint.PlainQuantity` – A `pint.PlainQuantity` wrapped numpy array.**default\_format:** `str = ''`

Default formatting string.

**property dimensionality:** `UnitsContainerT`

returns: `dict` – Dimensionality of the PlainQuantity, e.g. `{length: 1, time: -1}`

**property dimensionless:** `bool`

**dot**(*b*)

Dot product of two arrays.

Wraps `np.dot()`.

**fill**(*value*) → `None`

**property flat**

**property force\_ndarray:** `bool`

**property force\_ndarray\_like:** `bool`

**format\_babel**(*spec: str* = "", *\*\*kwspec: Any*) → `str`

**classmethod from\_list**(*quant\_list: list[PlainQuantity[MagnitudeT]]*, *units=None*) → `PlainQuantity[MagnitudeT]`

Transforms a list of Quantities into an `numpy.array` quantity. If no units are specified, the unit of the first element will be used. Same as `from_sequence`.

If units is not specified and list is empty, the unit cannot be determined and a `ValueError` is raised.

**Parameters**

- **quant\_list** (`list` of `pint.PlainQuantity`) – list of `pint.PlainQuantity`
- **units** (`UnitsContainer`, `str` or `pint.PlainQuantity`) – units of the physical quantity to be created (Default value = `None`)

**Returns** `pint.PlainQuantity`

**classmethod from\_sequence**(*seq: Sequence[PlainQuantity[MagnitudeT]]*, *units=None*) → `PlainQuantity[MagnitudeT]`

Transforms a sequence of Quantities into an `numpy.array` quantity. If no units are specified, the unit of the first element will be used.

If units is not specified and sequence is empty, the unit cannot be determined and a `ValueError` is raised.

**Parameters**

- **seq** (sequence of `pint.PlainQuantity`) – sequence of `pint.PlainQuantity`
- **units** (`UnitsContainer`, `str` or `pint.PlainQuantity`) – units of the physical quantity to be created (Default value = `None`)

**Returns** `pint.PlainQuantity`

**classmethod from\_tuple**(*tup*)

**property imag:** `pint.facets.numpy.quantity.NumpyQuantity`

**is\_compatible\_with**(*other: Any*, *\*contexts: Union[str, Context]*, *\*\*ctx\_kwargs: Any*) → `bool`

check if the other object is compatible

**Parameters**

- **other** – The object to check. Treated as dimensionless if not a `PlainQuantity`, `Unit` or `str`.
- **\*contexts** (`str` or `pint.Context`) – Contexts to use in the transformation.

- **\*\*ctx\_kwargs** – Values for the Context/s

Returns **bool**

**ito**(*other: Optional[QuantityOrUnitLike] = None, \*contexts, \*\*ctx\_kwargs*) → **None**

Inplace rescale to different units.

**Parameters**

- **other** (**pint.PlainQuantity**, **str** or **dict**) – Destination units. (Default value = **None**)
- **\*contexts** (**str** or **pint.Context**) – Contexts to use in the transformation.
- **\*\*ctx\_kwargs** – Values for the Context/s

**ito\_base\_units**() → **None**

Return PlainQuantity rescaled to plain units.

**ito\_preferred**(*preferred\_units: Optional[list[UnitLike]] = None*) → **PlainQuantity**

Return Quantity converted to a unit composed of the preferred units.

### Examples

```
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> (1*ureg.acre).to_preferred([ureg.meters])
<Quantity(4046.87261, 'meter ** 2')>
>>> (1*(ureg.force_pound*ureg.m)).to_preferred([ureg.W])
<Quantity(4.44822162, 'second * watt')>
```

**ito\_reduced\_units**() → **None**

Return PlainQuantity scaled in place to reduced units, i.e. one unit per dimension. This will not reduce compound units (e.g., 'J/kg' will not be reduced to  $m^2/s^2$ ), nor can it make use of contexts at this time.

**ito\_root\_units**() → **None**

Return PlainQuantity rescaled to root units.

**property m:** **pint.facets.plain.quantity.MagnitudeT**

PlainQuantity's magnitude. Short form for *magnitude*

**m\_as**(*units*) → **MagnitudeT**

PlainQuantity's magnitude expressed in particular units.

**Parameters** **units** (**pint.PlainQuantity**, **str** or **dict**) – destination units

**property magnitude:** **pint.facets.plain.quantity.MagnitudeT**

PlainQuantity's magnitude. Long form for *m*

**property ndim:** **int**

**persist**(*\*\*kwargs*)

Persist the Dask Array wrapped by pint.PlainQuantity.

**Parameters** **\*\*kwargs** (**dict**) – Any keyword arguments to pass to **dask.persist**.

**Returns** **pint.PlainQuantity** – A pint.PlainQuantity wrapped Dask array.

**plus\_minus**(*error, relative=False*)

**prod**(axis=None, dtype=None, out=None, keepdims=np.\_NoValue, initial=np.\_NoValue, where=np.\_NoValue)

Return the product of quantity elements over a given axis

Wraps np.prod().

**put**(indices, values, mode='raise') → None

**property real:** pint.facets.numpy.quantity.NumpyQuantity

**searchsorted**(v, side='left', sorter=None)

**property shape:** tuple[int, ...]

**to**(other: Optional[QuantityOrUnitLike] = None, \*contexts, \*\*ctx\_kwargs) → PlainQuantity

Return PlainQuantity rescaled to different units.

#### Parameters

- **other** (pint.PlainQuantity, str or dict) – destination units. (Default value = None)
- **\*contexts** (str or pint.Context) – Contexts to use in the transformation.
- **\*\*ctx\_kwargs** – Values for the Context/s

**Returns** pint.PlainQuantity

**to\_base\_units**() → PlainQuantity[MagnitudeT]

Return PlainQuantity rescaled to plain units.

**to\_compact**(unit: Optional[UnitsContainer] = None) → PlainQuantity

“Return PlainQuantity rescaled to compact, human-readable units.

To get output in terms of a different unit, use the unit parameter.

#### Examples

```
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> (200e-9*ureg.s).to_compact()
<Quantity(200.0, 'nanosecond')>
>>> (1e-2*ureg('kg m/s^2')).to_compact('N')
<Quantity(10.0, 'millinewton')>
```

**to\_openmm**() → openmm.unit.Quantity

Convert the quantity to an openmm.unit.Quantity.

**Returns** openmm\_quantity (openmm.unit.quantity.Quantity) – The OpenMM compatible quantity.

**to\_preferred**(preferred\_units: Optional[list[UnitLike]] = None) → PlainQuantity

Return Quantity converted to a unit composed of the preferred units.

## Examples

```
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> (1*ureg.acre).to_preferred([ureg.meters])
<Quantity(4046.87261, 'meter ** 2')>
>>> (1*(ureg.force_pound*ureg.m)).to_preferred([ureg.W])
<Quantity(4.44822162, 'second * watt')>
```

**to\_reduced\_units()** → PlainQuantity

Return PlainQuantity scaled in place to reduced units, i.e. one unit per dimension. This will not reduce compound units (intentionally), nor can it make use of contexts at this time.

**to\_root\_units()** → PlainQuantity[MagnitudeT]

Return PlainQuantity rescaled to root units.

**to\_timedelta()** → timedelta

**to\_tuple()** → tuple[MagnitudeT, tuple[tuple[str, ...]]]

**tolist()**

**property u:** Unit

PlainQuantity's units. Short form for *units*

**property unitless:** bool

**property units:** Unit

PlainQuantity's units. Long form for *u*

**visualize(\*\*kwargs)**

Produce a visual representation of the Dask graph.

The graphviz library is required.

**Parameters** **\*\*kwargs** (dict) – Any keyword arguments to pass to `dask.visualize`.

## Unit

**class** openff.units.Unit(\*args, \*\*kwargs)

Bases: Unit

## Methods

---

*compare*

---

*compatible\_units*

---

*format\_babel*

---

*from\_*

Converts a numerical value or quantity to this unit

*is\_compatible\_with*

check if the other object is compatible

*m\_from*

Converts a numerical value or quantity to this unit, then returns the magnitude of the converted value

---

## Attributes

<code>default_format</code>	Default formatting string.
<code>dimensionality</code>	returns: <code>dict</code> -- Dimensionality of the PlainUnit, e.g.
<code>dimensionless</code>	Return True if the PlainUnit is dimensionless; False otherwise.
<code>systems</code>	

---

**compare**(*other, op*) → `bool`

**compatible\_units**(\**contexts*)

**default\_format**: `str` = ''

Default formatting string.

**property dimensionality**: `UnitsContainer`

returns: `dict` – Dimensionality of the PlainUnit, e.g. {length: 1, time: -1}

**property dimensionless**: `bool`

Return True if the PlainUnit is dimensionless; False otherwise.

**format\_babel**(*spec=""*, *locale=None*, *\*\*kwspec: Any*) → `str`

**from\_**(*value*, *strict=True*, *name='value'*)

Converts a numerical value or quantity to this unit

### Parameters

- **value** – a Quantity (or numerical value if `strict=False`) to convert
- **strict** – boolean to indicate that only quantities are accepted (Default value = True)
- **name** – descriptive name to use if an exception occurs (Default value = “value”)

**Returns** `type` – The converted value as this unit

**is\_compatible\_with**(*other: Any*, *\*contexts: Union[str, Context]*, *\*\*ctx\_kwargs: Any*) → `bool`

check if the other object is compatible

### Parameters

- **other** – The object to check. Treated as dimensionless if not a Quantity, PlainUnit or str.
- **\*contexts** (`str` or `pint.Context`) – Contexts to use in the transformation.
- **\*\*ctx\_kwargs** – Values for the Context/s

**Returns** `bool`

**m\_from**(*value*, *strict=True*, *name='value'*)

Converts a numerical value or quantity to this unit, then returns the magnitude of the converted value

### Parameters

- **value** – a Quantity (or numerical value if `strict=False`) to convert
- **strict** – boolean to indicate that only quantities are accepted (Default value = True)
- **name** – descriptive name to use if an exception occurs (Default value = “value”)

**Returns** `type` – The magnitude of the converted value

## property systems

## Modules

<i>elements</i>	Symbols and masses for the chemical elements.
<i>exceptions</i>	
<i>openmm</i>	Functions for converting between OpenFF and OpenMM units
<i>utilities</i>	Utility methods for OpenFF Units

**elements**

Symbols and masses for the chemical elements.

This module provides mappings from atomic number to atomic mass and symbol. These dicts were seeded from running the below script using OpenMM 7.7.

It's not completely clear where OpenMM sourced these values from [1] but they are generally consistent with recent IUPAC values [2].

1. <https://github.com/openmm/openmm/issues/3434#issuecomment-1023406296>
2. <https://www.ciaaw.org/publications.htm>

```
import openmm.app

masses = {
    atomic_number: openmm.app.element.Element.getByAtomicNumber(
        atomic_number
    ).mass._value
    for atomic_number in range(1, 117)
}

symbols = {
    atomic_number: openmm.app.element.Element.getByAtomicNumber(atomic_number).symbol
    for atomic_number in range(1, 117)
}
```

## Module Attributes

<i>MASSES</i>	Mapping from atomic number to element symbol
<i>SYMBOLS</i>	Mapping from element symbol to atomic number

### MASSES

`openff.units.elements.MASSES: dict[int, Quantity]`

Mapping from atomic number to element symbol

### SYMBOLS

`openff.units.elements.SYMBOLS: dict[int, str]`

Mapping from element symbol to atomic number

### exceptions

#### Exceptions

<i>MissingOpenMMUnitError</i>	Raised when a unit cannot be converted to an equivalent OpenMM unit
<i>NoneQuantityError</i>	Raised when attempting to convert <i>None</i> between unit packages as a quantity object
<i>NoneUnitError</i>	Raised when attempting to convert <i>None</i> between unit packages as a unit object

#### MissingOpenMMUnitError

**exception** `openff.units.exceptions.MissingOpenMMUnitError`

Bases: `Exception`

Raised when a unit cannot be converted to an equivalent OpenMM unit

#### NoneQuantityError

**exception** `openff.units.exceptions.NoneQuantityError`

Bases: `Exception`

Raised when attempting to convert *None* between unit packages as a quantity object

#### NoneUnitError

**exception** `openff.units.exceptions.NoneUnitError`

Bases: `Exception`

Raised when attempting to convert *None* between unit packages as a unit object



## openmm

Functions for converting between OpenFF and OpenMM units

### Functions

<code>from_openmm</code>	Convert an OpenMM Quantity to an OpenFF Quantity
<code>to_openmm</code>	Convert an OpenFF Quantity to an OpenMM Quantity
<code>openmm_unit_to_string</code>	Convert a openmm.unit.Unit to a string representation.
<code>string_to_openmm_unit</code>	Deserializes a openmm.unit.Quantity from a string representation, for example: "kilocalories_per_mole / angstrom ** 2"
<code>ensure_quantity</code>	Given a quantity that could be of a variety of types, attempt to coerce into a given type.

## from\_openmm

`openff.units.openmm.from_openmm(openmm_quantity: openmm.unit.Quantity) → Quantity`

Convert an OpenMM Quantity to an OpenFF Quantity

`openmm.unit.quantity.Quantity` from OpenMM and `openff.units.Quantity` from this package both represent a numerical value with units.

### Examples

```

>>> from openff.units import Quantity as OpenFFQuantity
>>> from openff.units.openmm import from_openmm
>>> from openmm import unit
>>> length = unit.Quantity(9.0, unit.angstrom)
>>> from_openmm(length)
<Quantity(9.0, 'angstrom')>
>>> assert isinstance(from_openmm(length), OpenFFQuantity)

```

## to\_openmm

`openff.units.openmm.to_openmm(quantity: Quantity) → openmm.unit.Quantity`

Convert an OpenFF Quantity to an OpenMM Quantity

`openmm.unit.quantity.Quantity` from OpenMM and `openff.units.Quantity` from this package both represent a numerical value with units. The units available in the two packages differ; when a unit is missing from the target package, the resulting quantity will be in base units (kg/m/s/A/K/mole), which are shared between both packages. This may cause the resulting value to be slightly different to the input due to the limited precision of floating point numbers.

## Examples

```

>>> from openff.units import unit
>>> from openff.units.openmm import to_openmm
>>> from openmm import unit as openmm_unit
>>> length = unit.Quantity(9.0, unit.angstrom)
>>> to_openmm(length)
Quantity(value=9.0, unit=angstrom)
>>> assert isinstance(to_openmm(length), openmm_unit.Quantity)

```

`openmm_unit_to_string`

`openff.units.openmm.openmm_unit_to_string(input_unit: openmm.unit.Unit) → str`

Convert a openmm.unit.Unit to a string representation.

**Parameters** `input_unit` (A openmm.unit) – The unit to serialize

**Returns** `unit_string` (str) – The serialized unit.

`string_to_openmm_unit`

`openff.units.openmm.string_to_openmm_unit(unit_string: str) → openmm.unit.Unit`

Deserializes a openmm.unit.Quantity from a string representation, for example: “kilocalories\_per\_mole / angstrom \*\* 2”

**Parameters** `unit_string` (dict) – Serialized representation of a openmm.unit.Quantity.

**Returns** `output_unit` (openmm.unit.Quantity) – The deserialized unit from the string

**Raises** `MissingOpenMMUnitError` – if the unit is unavailable in OpenMM.

`ensure_quantity`

`openff.units.openmm.ensure_quantity(unknown_quantity: Quantity, type_to_ensure: Literal['openmm', 'openff']) → Quantity`

Given a quantity that could be of a variety of types, attempt to coerce into a given type.

## Examples

```

>>> import numpy
>>> from openmm import unit as openmm_unit
>>> from openff.units import unit
>>> from openff.units.openmm import ensure_quantity
>>> # Create a 9 Angstrom quantity with each registry
>>> length1 = unit.Quantity(9.0, unit.angstrom)
>>> length2 = openmm_unit.Quantity(9.0, openmm_unit.angstrom)
>>> # Similar quantities are be coerced into requested type
>>> assert type(ensure_quantity(length1, "openmm")) == openmm_unit.Quantity
>>> assert type(ensure_quantity(length2, "openff")) == unit.Quantity
>>> # Seemingly-redundant "conversions" short-circuit

```

(continues on next page)

(continued from previous page)

```

>>> assert ensure_quantity(length1, "openff") == ensure_quantity(length2, "openff")
>>> assert ensure_quantity(length1, "openmm") == ensure_quantity(length2, "openmm")
>>> # NumPy arrays and some primitives are automatically up-converted to `Quantity`
    ↳ objects
>>> # Note that their units are set to "dimensionless"
>>> ensure_quantity(numpy.array([1, 2]), "openff")
<Quantity([1 2], 'dimensionless')>
>>> ensure_quantity(4.0, "openmm")
Quantity(value=4.0, unit=dimensionless)

```

## utilities

Utility methods for OpenFF Units

## Functions

---

`get_defaults_path`

Get the full path to the `defaults.txt` file

---

### `get_defaults_path`

`openff.units.utilities.get_defaults_path()` → str

Get the full path to the `defaults.txt` file



## PYTHON MODULE INDEX

### O

- `openff.units`, [9](#)
- `openff.units.elements`, [19](#)
- `openff.units.exceptions`, [20](#)
- `openff.units.openmm`, [21](#)
- `openff.units.utilities`, [23](#)



## C

check() (*openff.units.Quantity* method), 13  
clip() (*openff.units.Quantity* method), 13  
compare() (*openff.units.Quantity* method), 13  
compare() (*openff.units.Unit* method), 18  
compatible\_units() (*openff.units.Quantity* method), 13  
compatible\_units() (*openff.units.Unit* method), 18  
compute() (*openff.units.Quantity* method), 13

## D

default\_format (*openff.units.Quantity* attribute), 13  
default\_format (*openff.units.Unit* attribute), 18  
dimensionality (*openff.units.Quantity* property), 13  
dimensionality (*openff.units.Unit* property), 18  
dimensionless (*openff.units.Quantity* property), 14  
dimensionless (*openff.units.Unit* property), 18  
dot() (*openff.units.Quantity* method), 14

## E

ensure\_quantity() (in module *openff.units*), 10  
ensure\_quantity() (in module *openff.units.openmm*), 22

## F

fill() (*openff.units.Quantity* method), 14  
flat (*openff.units.Quantity* property), 14  
force\_ndarray (*openff.units.Quantity* property), 14  
force\_ndarray\_like (*openff.units.Quantity* property), 14  
format\_babel() (*openff.units.Quantity* method), 14  
format\_babel() (*openff.units.Unit* method), 18  
from\_() (*openff.units.Unit* method), 18  
from\_list() (*openff.units.Quantity* class method), 14  
from\_openmm() (in module *openff.units.openmm*), 21  
from\_sequence() (*openff.units.Quantity* class method), 14  
from\_tuple() (*openff.units.Quantity* class method), 14

## G

get\_defaults\_path() (in module *openff.units.utilities*), 23

## I

imag (*openff.units.Quantity* property), 14  
is\_compatible\_with() (*openff.units.Quantity* method), 14  
is\_compatible\_with() (*openff.units.Unit* method), 18  
ito() (*openff.units.Quantity* method), 15  
ito\_base\_units() (*openff.units.Quantity* method), 15  
ito\_preferred() (*openff.units.Quantity* method), 15  
ito\_reduced\_units() (*openff.units.Quantity* method), 15  
ito\_root\_units() (*openff.units.Quantity* method), 15

## M

m (*openff.units.Quantity* property), 15  
m\_as() (*openff.units.Quantity* method), 15  
m\_from() (*openff.units.Unit* method), 18  
magnitude (*openff.units.Quantity* property), 15  
MASSES (in module *openff.units.elements*), 20  
Measurement() (in module *openff.units*), 10  
MissingOpenMMUnitError, 20  
module  
    *openff.units*, 9  
    *openff.units.elements*, 19  
    *openff.units.exceptions*, 20  
    *openff.units.openmm*, 21  
    *openff.units.utilities*, 23

## N

ndim (*openff.units.Quantity* property), 15  
NoneQuantityError, 20  
NoneUnitError, 20

## O

*openff.units*  
    module, 9  
*openff.units.elements*  
    module, 19  
*openff.units.exceptions*  
    module, 20  
*openff.units.openmm*  
    module, 21  
*openff.units.utilities*

module, 23  
openmm\_unit\_to\_string() (in module  
openff.units.openmm), 22

## P

persist() (openff.units.Quantity method), 15  
plus\_minus() (openff.units.Quantity method), 15  
prod() (openff.units.Quantity method), 15  
put() (openff.units.Quantity method), 16

## Q

Quantity (class in openff.units), 11

## R

real (openff.units.Quantity property), 16

## S

searchsorted() (openff.units.Quantity method), 16  
shape (openff.units.Quantity property), 16  
string\_to\_openmm\_unit() (in module  
openff.units.openmm), 22  
SYMBOLS (in module openff.units.elements), 20  
systems (openff.units.Unit property), 18

## T

T (openff.units.Quantity property), 13  
to() (openff.units.Quantity method), 16  
to\_base\_units() (openff.units.Quantity method), 16  
to\_compact() (openff.units.Quantity method), 16  
to\_openmm() (in module openff.units.openmm), 21  
to\_openmm() (openff.units.Quantity method), 16  
to\_preferred() (openff.units.Quantity method), 16  
to\_reduced\_units() (openff.units.Quantity method),  
17  
to\_root\_units() (openff.units.Quantity method), 17  
to\_timedelta() (openff.units.Quantity method), 17  
to\_tuple() (openff.units.Quantity method), 17  
tolist() (openff.units.Quantity method), 17

## U

u (openff.units.Quantity property), 17  
Unit (class in openff.units), 17  
unit (in module openff.units), 9  
unitless (openff.units.Quantity property), 17  
units (openff.units.Quantity property), 17  
UnitsContainer (openff.units.Quantity property), 13

## V

visualize() (openff.units.Quantity method), 17