
OpenSCM-Units Documentation

Release 0.2.0+28.g284aa05.dirty

Zeb Nicholls, Sven Willner, Jared Lewis, Robert Gieseke

Jan 08, 2021

DOCUMENTATION

1	Installation	3
2	Usage	5
3	Development	7
3.1	Contributing	7
3.2	Getting setup	8
3.3	Formatting	9
3.4	Buiding the docs	10
3.5	Releasing	10
3.6	Why is there a <code>Makefile</code> in a pure Python repository?	11
4	Unit Registry API	13
5	Data API	23
5.1	Mixtures API	23
6	Changelog	25
6.1	master	25
6.2	v0.2.0	25
6.3	v0.1.4	25
6.4	v0.1.3	25
6.5	v0.1.2	26
6.6	v0.1.1	26
6.7	v0.1.0	26
7	Index	27
	Python Module Index	29
	Index	31

OpenSCM-Units is a repository for handling of units related to simple climate modelling.

OpenSCM-Units is free software under a BSD 3-Clause License, see [LICENSE](#).

INSTALLATION

OpenSCM-Runner can be installed with pip

```
pip install openscm-units
```

If you also want to run the example notebooks install additional dependencies using

```
pip install openscm-units[notebooks]
```

OpenSCM-Units can also be installed with conda

```
conda install -c conda-forge openscm-units
```

CHAPTER TWO

USAGE

All of our usage examples are included in `openscm-units/notebooks`.

DEVELOPMENT

If you're interested in contributing to OpenSCM-Units, we'd love to have you on board! This section of the docs will (once we've written it) detail how to get setup to contribute and how best to communicate.

- *Contributing*
- *Getting setup*
 - *Getting help*
 - * *Development tools*
 - * *Other tools*
- *Formatting*
- *Buiding the docs*
 - *Gotchas*
 - *Docstring style*
- *Releasing*
 - *First step*
 - *PyPI*
 - *Push to repository*
- *Why is there a Makefile in a pure Python repository?*

3.1 Contributing

All contributions are welcome, some possible suggestions include:

- tutorials (or support questions which, once solved, result in a new tutorial :D)
- blog posts
- improving the documentation
- bug reports
- feature requests
- pull requests

Please report issues or discuss feature requests in the [OpenSCM-Units issue tracker](#). If your issue is a feature request or a bug, please use the templates available, otherwise, simply open a normal issue :)

As a contributor, please follow a couple of conventions:

- Create issues in the [OpenSCM-Units issue tracker](#) for changes and enhancements, this ensures that everyone in the community has a chance to comment
- Be welcoming to newcomers and encourage diverse new contributors from all backgrounds: see the [Python Community Code of Conduct](#)
- Only push to your own branches, this allows people to force push to their own branches as they need without fear or causing others headaches
- Start all pull requests as draft pull requests and only mark them as ready for review once they've been rebased onto master, this makes it much simpler for reviewers
- Try and make lots of small pull requests, this makes it easier for reviewers and faster for everyone as review time grows exponentially with the number of lines in a pull request

3.2 Getting setup

To get setup as a developer, we recommend the following steps (if any of these tools are unfamiliar, please see the resources we recommend in [Development tools](#)):

1. Install conda and make
2. Run `make virtual-environment`, if that fails you can try doing it manually
 1. Change your current directory to OpenSCM-Units's root directory (i.e. the one which contains `README.rst`), `cd openscm-units`
 2. Create a virtual environment to use with OpenSCM-Units `python3 -m venv venv`
 3. Activate your virtual environment `source ./venv/bin/activate`
 4. Upgrade pip `pip install --upgrade pip`
 5. Install the development dependencies (very important, make sure your virtual environment is active before doing this) `pip install -e .[dev]`
3. Make sure the tests pass by running `make test-all`, if that fails the commands are
 1. Activate your virtual environment `source ./venv/bin/activate`
 2. Run the unit and integration tests `pytest --cov -r a --cov-report term-missing`
 3. Test the notebooks `pytest -r a --nbval ./notebooks --sanitize ./notebooks/tests_sanitize.cfg`

3.2.1 Getting help

Whilst developing, unexpected things can go wrong (that's why it's called 'developing', if we knew what we were doing, it would already be 'developed'). Normally, the fastest way to solve an issue is to contact us via the [issue tracker](#). The other option is to debug yourself. For this purpose, we provide a list of the tools we use during our development as starting points for your search to find what has gone wrong.

Development tools

This list of development tools is what we rely on to develop OpenSCM-Units reliably and reproducibly. It gives you a few starting points in case things do go inexplicably wrong and you want to work out why. We include links with each of these tools to starting points that we think are useful, in case you want to learn more.

- [Git](#)
- [Make](#)
- [Conda virtual environments](#)
- [Pip and pip virtual environments](#)
- [Tests](#)
 - we use a blend of [pytest](#) and the inbuilt Python testing capabilities for our tests so checkout what we've already done in `tests` to get a feel for how it works
- [Continuous integration \(CI\)](#) (also [brief intro blog post](#) and a [longer read](#))
 - we use GitHub CI for our CI but there are a number of good providers
- [Jupyter Notebooks](#)
 - Jupyter is automatically included in your virtual environment if you follow our [Getting setup](#) instructions
- [Sphinx](#)

Other tools

We also use some other tools which aren't necessarily the most familiar. Here we provide a list of these along with useful resources.

- [Regular expressions](#)
 - we use [regex101.com](#) to help us write and check our regular expressions, make sure the language is set to Python to make your life easy!

3.3 Formatting

To help us focus on what the code does, not how it looks, we use a couple of automatic formatting tools. These automatically format the code for us and tell use where the errors are. To use them, after setting yourself up (see [Getting setup](#)), simply run `make format` (and `make format-notebooks` to format notebook code). Note that `make format` can only be run if you have committed all your work i.e. your working directory is 'clean'. This restriction is made to ensure that you don't format code without being able to undo it, just in case something goes wrong.

3.4 Buiding the docs

After setting yourself up (see [Getting setup](#)), building the docs is as simple as running `make docs` (note, run `make -B docs` to force the docs to rebuild and ignore make when it says ‘... index.html is up to date’). This will build the docs for you. You can preview them by opening `docs/build/html/index.html` in a browser.

For documentation we use [Sphinx](#). To get ourselves started with Sphinx, we started with [this example](#) then used [Sphinx’s getting started guide](#).

3.4.1 Gotchas

To get Sphinx to generate pdfs (rarely worth the hassle), you require [Latexmk](#). On a Mac this can be installed with `sudo tlmgr install latexmk`. You will most likely also need to install some other packages (if you don’t have the full distribution). You can check which package contains any missing files with `tlmgr search --global --file [filename]`. You can then install the packages with `sudo tlmgr install [package]`.

3.4.2 Docstring style

For our docstrings we use numpy style docstrings. For more information on these, [here is the full guide](#) and [the quick reference we also use](#).

3.5 Releasing

3.5.1 First step

1. Test installation with dependencies `make test-install`
2. Update `CHANGELOG.rst`
 - add a header for the new version between master and the latest bullet point
 - this should leave the section underneath the master header empty
3. `git add .`
4. `git commit -m "Prepare for release of vX.Y.Z"`
5. `git tag vX.Y.Z`
6. Test version updated as intended with `make test-install`

3.5.2 PyPI

If uploading to PyPI, do the following (otherwise skip these steps)

1. `make publish-on-testpypi`
2. Go to [test PyPI](#) and check that the new release is as intended. If it isn’t, stop and debug.
3. Test the install with `make test-testpypi-install` (this doesn’t test all the imports as most required packages are not on test PyPI).

Assuming test PyPI worked, now upload to the main repository

1. `make publish-on-pypi`
2. Go to [OpenSCM-Units's PyPI](#) and check that the new release is as intended.
3. Test the install with `make test-pypi-install`

3.5.3 Push to repository

Finally, push the tags and the repository state

1. `git push`
2. `git push --tags`

3.6 Why is there a `Makefile` in a pure Python repository?

Whilst it may not be standard practice, a `Makefile` is a simple way to automate general setup (environment setup in particular). Hence we have one here which basically acts as a notes file for how to do all those little jobs which we often forget e.g. setting up environments, running tests (and making sure we're in the right environment), building docs, setting up auxillary bits and pieces.

UNIT REGISTRY API

Unit handling makes use of the [Pint](#) library. This allows us to easily define units as well as contexts. Contexts allow us to perform conversions which would not normally be allowed e.g. in the ‘AR4GWP100’ context we can convert from CO2 to CH4 using the AR4GWP100 equivalence metric.

An illustration of how the `unit_registry` can be used is shown below:

```
>>> from openscm_units import unit_registry
>>> unit_registry("CO2")
<Quantity(1, 'CO2')>

>>> emissions_aus = 0.34 * unit_registry("Gt C / yr")
>>> emissions_aus
<Quantity(0.34, 'C * gigametric_ton / a')>

>>> emissions_aus.to("Mt CO2 / yr")
<Quantity(1246.6666666666667, 'CO2 * megametric_ton / a')>

>>> with unit_registry.context("AR4GWP100"):
...     (100 * unit_registry("Mt CH4 / yr")).to("Mt CO2 / yr")
<Quantity(2500.0, 'CO2 * megametric_ton / a')>
```

More details on emissions units

Emissions are a flux composed of three parts: mass, the species being emitted and the time period e.g. “t CO2 / yr”. As mass and time are part of SI units, all we need to define here are emissions units i.e. the stuff. Here we include as many of the canonical emissions units, and their conversions, as possible.

For emissions units, there are a few cases to be considered:

- fairly obvious ones e.g. carbon dioxide emissions can be provided in ‘C’ or ‘CO2’ and converting between the two is possible
- less obvious ones e.g. nitrous oxide emissions can be provided in ‘N’, ‘N2O’ or ‘N2ON’ (a short-hand which indicates that only the mass of the nitrogen is being counted), we provide conversions between these three
- case-sensitivity. In order to provide a simplified interface, using all uppercase versions of any unit is also valid e.g. `unit_registry("HFC4310mee")` is the same as `unit_registry("HFC4310MEE")`
- hyphens and underscores in units. In order to be Pint compatible and to simplify things, we strip all hyphens and underscores from units.

As a convenience, we allow users to combine the mass and the type of emissions to make a ‘joint unit’ e.g. “tCO2”. It should be recognised that this joint unit is a derived unit and not a base unit.

By defining these three separate components, it is much easier to track what conversions are valid and which are not. For example, as the emissions units are all defined as emissions units, and not as atomic masses, we are able to prevent invalid conversions. If emissions units were simply atomic masses, it would be possible to convert between e.g. C and

N₂O which would be a problem. Conventions such as allowing carbon dioxide emissions to be reported in C or CO₂, despite the fact that they are fundamentally different chemical species, is a convention which is particular to emissions (as far as we can tell).

Pint's contexts are particularly useful for emissions as they facilitate metric conversions. With a context, a conversion which wouldn't normally be allowed (e.g. tCO₂ → tN₂O) is allowed and will use whatever metric conversion is appropriate for that context (e.g. AR4GWP100).

Finally, we discuss namespace collisions.

CH₄

Methane emissions are defined as 'CH₄'. In order to prevent inadvertent conversions of 'CH₄' to e.g. 'CO₂' via 'C', the conversion 'CH₄' ↔ 'C' is by default forbidden. However, it can be performed within the context 'CH₄_conversions' as shown below:

```
>>> from openscm_units import unit_registry
>>> unit_registry("CH4").to("C")
pint.errors.DimensionalityError: Cannot convert from 'CH4' ([methane]) to 'C'
↳ ([carbon])

# with a context, the conversion becomes legal again
>>> with unit_registry.context("CH4_conversions"):
...     unit_registry("CH4").to("C")
<Quantity(0.75, 'C')>

# as an unavoidable side effect, this also becomes possible
>>> with unit_registry.context("CH4_conversions"):
...     unit_registry("CH4").to("CO2")
<Quantity(2.75, 'CO2')>
```

NO_x

Like for methane, NO_x emissions also suffer from a namespace collision. In order to prevent inadvertent conversions from 'NO_x' to e.g. 'N₂O', the conversion 'NO_x' ↔ 'N' is by default forbidden. It can be performed within the 'NO_x_conversions' context:

```
>>> from openscm_units import unit_registry
>>> unit_registry("NOx").to("N")
pint.errors.DimensionalityError: Cannot convert from 'NOx' ([NOx]) to 'N' ([nitrogen])

# with a context, the conversion becomes legal again
>>> with unit_registry.context("NOx_conversions"):
...     unit_registry("NOx").to("N")
<Quantity(0.30434782608695654, 'N')>

# as an unavoidable side effect, this also becomes possible
>>> with unit_registry.context("NOx_conversions"):
...     unit_registry("NOx").to("N2O")
<Quantity(0.9565217391304348, 'N2O')>
```

NH₃

In order to prevent inadvertent conversions from 'NH₃' to 'CO₂', the conversion 'NH₃' ↔ 'N' is by default forbidden. It can be performed within the 'NH₃_conversions' context analogous to the 'NO_x_conversions' context:

```
>>> from openscm_units import unit_registry
>>> unit_registry("NH3").to("N")
pint.errors.DimensionalityError: Cannot convert from 'NH3' ([NH3]) to 'N' ([nitrogen])
```

(continues on next page)

(continued from previous page)

```
# with a context, the conversion becomes legal again
>>> with unit_registry.context("NH3_conversions"):
...     unit_registry("NH3").to("N")
<Quantity(0.823529412, 'N')>
```

class openscm_units._unit_registry.**ScmUnitRegistry** (*args, **kwargs)

Bases: pint.registry.UnitRegistry

Unit registry class.

Provides some convenience methods to add standard units and contexts with lazy loading from disk.

UnitsContainer (*args, **kwargs)

add_context (context: pint.context.Context) → None

Add a context object to the registry.

The context will be accessible by its name and aliases.

Notice that this method will NOT enable the context; see `enable_contexts()`.

add_standards ()

Add standard units.

Has to be done separately because of pint's weird initializing.

auto_reduce_dimensions

Determines if dimensionality should be reduced on appropriate operations.

case_sensitive

Default unit case sensitivity

check (*args)

Decorator to for quantity type checking for function inputs.

Use it to ensure that the decorated function input parameters match the expected dimension of pint quantity.

The wrapper function raises:

- `pint.DimensionalityError` if an argument doesn't match the required dimensions.

ureg [UnitRegistry] a UnitRegistry instance.

args [str or UnitContainer or None] Dimensions of each of the input arguments. Use `None` to skip argument conversion.

Returns the wrapped function.

Return type callable

Raises

- **TypeError** – If the number of given dimensions does not match the number of function parameters.
- **ValueError** – If the any of the provided dimensions cannot be parsed as a dimension.

context (*names, **kwargs)

Used as a context manager, this function enables to activate a context which is removed after usage.

Parameters

- ***names** – name(s) of the context(s).

- ****kwargs** – keyword arguments for the contexts.

Examples

Context can be called by their name:

```
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> ureg.add_context(pint.Context('one'))
>>> ureg.add_context(pint.Context('two'))
>>> with ureg.context('one'):
...     pass
```

If a context has an argument, you can specify its value as a keyword argument:

```
>>> with ureg.context('one', n=1):
...     pass
```

Multiple contexts can be entered in single call:

```
>>> with ureg.context('one', 'two', n=1):
...     pass
```

Or nested allowing you to give different values to the same keyword argument:

```
>>> with ureg.context('one', n=1):
...     with ureg.context('two', n=2):
...         pass
```

A nested context inherits the defaults from the containing context:

```
>>> with ureg.context('one', n=1):
...     # Here n takes the value of the outer context
...     with ureg.context('two'):
...         pass
```

convert (*value*, *src*, *dst*, *inplace=False*)

Convert value from some source to destination units.

Parameters

- **value** – value
- **src** (*pint.Quantity* or *str*) – source units.
- **dst** (*pint.Quantity* or *str*) – destination units.
- **inplace** – (Default value = False)

Returns converted value

Return type *type*

default_as_delta

When performing a multiplication of units, interpret non-multiplicative units as their *delta* counterparts.

property default_format

Default formatting string for quantities.

property default_system

define (*definition*)

Add unit to the registry.

Parameters **definition** (*str* or *Definition*) – a dimension, unit or prefix definition.

disable_contexts (*n*: *Optional[int]* = *None*) → *None*

Disable the last *n* enabled contexts.

Parameters **n** (*int*) – Number of contexts to disable. Default: disable all contexts.

enable_contexts (**names_or_contexts*, ***kwargs*)

Overload pint's *enable_contexts()* to load contexts once (the first time they are used) to avoid (unnecessary) file operations on import.

fmt_locale = *None*

get_base_units (*input_units*, *check_nonmult=True*, *system=None*)

Convert unit or dict of units to the base units.

If any unit is non multiplicative and *check_converter* is *True*, then *None* is returned as the multiplicative factor.

Unlike BaseRegistry, in this registry *root_units* might be different from *base_units*

Parameters

- **input_units** (*UnitsContainer* or *str*) – units
- **check_nonmult** (*bool*) – if *True*, *None* will be returned as the multiplicative factor if a non-multiplicative units is found in the final Units. (Default value = *True*)
- **system** – (Default value = *None*)

Returns multiplicative factor, base units

Return type *type*

get_compatible_units (*input_units*, *group_or_system=None*)

get_dimensionality (*input_units*)

Convert unit or dict of units or dimensions to a dict of base dimensions dimensions

get_group (*name*, *create_if_needed=True*)

Return a Group.

Parameters

- **name** (*str*) – Name of the group to be
- **create_if_needed** (*bool*) – If *True*, create a group if not found. If *False*, raise an Exception. (Default value = *True*)

Returns Group

Return type *type*

get_name (*name_or_alias*, *case_sensitive=None*)

Return the canonical name of a unit.

get_root_units (*input_units*, *check_nonmult=True*)

Convert unit or dict of units to the root units.

If any unit is non multiplicative and *check_converter* is *True*, then *None* is returned as the multiplicative factor.

Parameters

- **input_units** (*UnitsContainer* or *str*) – units
- **check_nonmult** (*bool*) – if True, None will be returned as the multiplicative factor if a non-multiplicative units is found in the final Units. (Default value = True)

Returns multiplicative factor, base units

Return type Number, pint.Unit

get_symbol (*name_or_alias*, *case_sensitive=None*)

Return the preferred alias for a unit.

get_system (*name*, *create_if_needed=True*)

Return a Group.

Parameters

- **name** (*str*) – Name of the group to be
- **create_if_needed** (*bool*) – If True, create a group if not found. If False, raise an Exception. (Default value = True)

Returns System

Return type *type*

is_compatible_with (*obj1*, *obj2*, **contexts*, ***ctx_kwargs*)

check if the other object is compatible

Parameters

- **obj1** – The objects to check against each other. Treated as dimensionless if not a Quantity, Unit or str.
- **obj2** – The objects to check against each other. Treated as dimensionless if not a Quantity, Unit or str.
- ***contexts** (*str* or *pint.Context*) – Contexts to use in the transformation.
- ****ctx_kwargs** – Values for the Context/s

Returns

Return type *bool*

load_definitions (*file*, *is_resource=False*)

Add units and prefixes defined in a definition text file.

Parameters

- **file** – can be a filename or a line iterable.
- **is_resource** – used to indicate that the file is a resource file and therefore should be loaded from the package. (Default value = False)

non_int_type

Numerical type used for non integer values.

parse_expression (*input_string*, *case_sensitive=None*, *use_decimal=False*, ***values*)

Parse a mathematical expression including units and return a quantity object.

Numerical constants can be specified as keyword arguments and will take precedence over the names defined in the registry.

Parameters

- **input_string** –

- **case_sensitive** – (Default value = None, which uses registry setting)
- **use_decimal** – (Default value = False)
- ****values** –

parse_pattern (*input_string*, *pattern*, *case_sensitive=None*, *use_decimal=False*, *many=False*)

Parse a string with a given regex pattern and returns result.

Parameters

- **input_string** –
- **pattern_string** – The regex parse string
- **case_sensitive** – (Default value = None, which uses registry setting)
- **use_decimal** – (Default value = False)
- **many** – Match many results (Default value = False)

parse_unit_name (*unit_name*, *case_sensitive=None*)

Parse a unit to identify prefix, unit name and suffix by walking the list of prefix and suffix. In case of equivalent combinations (e.g. ('kilo', 'gram', '') and ('', 'kilogram', '')), prefer those with prefix.

Parameters

- **unit_name** –
- **case_sensitive** (*bool* or *None*) – Control if unit lookup is case sensitive. Defaults to None, which uses the registry's case_sensitive setting

Returns all non-equivalent combinations of (prefix, unit name, suffix)

Return type tuple of tuples (*str*, *str*, *str*)

parse_units (*input_string*, *as_delta=None*, *case_sensitive=None*)

Parse a units expression and returns a UnitContainer with the canonical names.

The expression can only contain products, ratios and powers of units.

Parameters

- **input_string** (*str*) –
- **as_delta** (*bool* or *None*) – if the expression has multiple units, the parser will interpret non multiplicative units as their *delta_* counterparts. (Default value = None)
- **case_sensitive** (*bool* or *None*) – Control if unit parsing is case sensitive. Defaults to None, which uses the registry's setting.

pi_theorem (*quantities*)

Builds dimensionless quantities using the Buckingham theorem

Parameters **quantities** (*dict*) – mapping between variable name and units

Returns a list of dimensionless quantities expressed as dicts

Return type *list*

remove_context (*name_or_alias: str*) → *pint.context.Context*

Remove a context from the registry and return it.

Notice that this methods will not disable the context; see *disable_contexts()*.

set_fmt_locale (*loc*)

Change the locale used by default by *format_babel*.

Parameters `loc` (*str* or *None*) – *None*` (do not translate), ‘sys’ (detect the system locale) or a locale id string.

setup_matplotlib (*enable=True*)

Set up handlers for matplotlib’s unit support.

Parameters `enable` (*bool*) – whether support should be enabled or disabled (Default value = *True*)

split_gas_mixture (*quantity: pint.quantity.Quantity*) → *list*

Split a gas mixture into constituent gases.

Given a pint quantity with the units containing a gas mixture, returns a list of the constituents as pint quantities.

property sys

with_context (*name, **kwargs*)

Decorator to wrap a function call in a Pint context.

Use it to ensure that a certain context is active when calling a function:

```
:param name: name of the context.  
:param \**kwargs: keyword arguments for the context
```

Returns the wrapped function.

Return type callable

Example

```
>>> @ureg.with_context('sp')  
... def my_cool_fun(wavelength):  
...     print('This wavelength is equivalent to: %s', wavelength.to('terahertz'  
↵'))
```

wraps (*ret, args, strict=True*)

Wraps a function to become pint-aware.

Use it when a function requires a numerical value but in some specific units. The wrapper function will take a pint quantity, convert to the units specified in *args* and then call the wrapped function with the resulting magnitude.

The value returned by the wrapped function will be converted to the units specified in *ret*.

Parameters

- **ureg** (*pint.UnitRegistry*) – a UnitRegistry instance.
- **ret** (*str, pint.Unit, iterable of str, or iterable of pint.Unit*) – Units of each of the return values. Use *None* to skip argument conversion.
- **args** (*str, pint.Unit, iterable of str, or iterable of pint.Unit*) – Units of each of the input arguments. Use *None* to skip argument conversion.
- **strict** (*bool*) – Indicates that only quantities are accepted. (Default value = *True*)

Returns the wrapper function.

Return type callable

Raises `TypeError` – if the number of given arguments does not match the number of function parameters. if the any of the provided arguments is not a unit a string or Quantity

```
openscm_units._unit_registry.unit_registry = <openscm_units._unit_registry.ScmUnitRegistry
Standard unit registry
```

The unit registry contains all of the recognised units. Be careful, if you edit this registry in one place then it will also be edited in any other places that use `openscm_units`. If you want multiple, separate registries, create multiple instances of `ScmUnitRegistry`.

DATA API

Data used within OpenSCM Units

For example, metric conversions and breakdowns of mixture substances into their constituents.

5.1 Mixtures API

`openscm_units.data.mixtures`.**MIXTURES**

Gas mixtures supported by OpenSCM Units

Last update: 2020-12-16

Each key is the mixture's name. Each value is itself a dictionary where each key is the name of a component of the mixture and the value is a list in which the first element is the standard composition, the second element is the positive composition tolerance and the third element is the negative composition tolerance. All values are given in mass percentage.

Sources:

- ANSI/ASHRAE Standard 34-2019, p. 9ff, ISSN 1041-2336, https://www.techstreet.com/ashrae/standards/ashrae-15-2019-packaged-w-34-2019?product_id=2046531
- https://en.wikipedia.org/wiki/List_of_refrigerants (for common names)

Type dict

CHANGELOG

6.1 master

- (#20) Make `openscm_units.data` a module by adding an `__init__.py` file to it and add docs for `openscm_units.data` (closes #19)
- (#18) Made NH3 a separate dimension to avoid accidental conversion to CO2 in GWP contexts. Also added an `nh3_conversions` context to convert to nitrogen (closes #12)
- (#16) Added refrigerant mixtures as units, including automatic GWP calculation from the GWP of their constituents. Also added the `unit_registry.split_gas_mixtures` function which can be used to split quantities containing a gas mixture into their constituents (closes #10)

6.2 v0.2.0

- (#15) Update CI so that it runs on pull requests from forks too
- (#14) Renamed `openscm_units.unit_registry` module to `openscm_units._unit_registry` to avoid name collision and lift `ScmUnitRegistry` to `openscm_units.ScmUnitRegistry` (closes #13)

6.3 v0.1.4

- (#7) Added C7F16, C8F18 and SO2F2 AR5GWP100 (closes #8)

6.4 v0.1.3

- (#7) Include metric conversions data in package
- (#6) Add conda install instructions

6.5 v0.1.2

- (#5) Update `MANIFEST.in` to ensure `LICENSE`, `README.rst` and `CHANGELOG.rst` are included in source distributions
- (#4) Update `README` and url to point to openscm organisation

6.6 v0.1.1

- (#2) Hotfix so that 'Tt' is terra tonne rather than 'tex'

6.7 v0.1.0

- (#1) Setup repository

INDEX

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

`openscm_units._unit_registry`, [13](#)

`openscm_units.data`, [23](#)

INDEX

A

`add_context()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 15

`add_standards()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 15

`auto_reduce_dimensions` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 attribute), 15

C

`case_sensitive` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 attribute), 15

`check()` (`openscm_units._unit_registry.ScmUnitRegistry`
 method), 15

`context()` (`openscm_units._unit_registry.ScmUnitRegistry`
 method), 15

`convert()` (`openscm_units._unit_registry.ScmUnitRegistry`
 method), 16

D

`default_as_delta` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 attribute), 16

`default_format()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 property), 16

`default_system()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 property), 16

`define()` (`openscm_units._unit_registry.ScmUnitRegistry`
 method), 16

`disable_contexts()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 17

E

`enable_contexts()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 17

F

`fmt_locale` (`openscm_units._unit_registry.ScmUnitRegistry`
 attribute), 17

G

`get_base_units()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 17

`get_compatible_units()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 17

`get_dimensionality()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 17

`get_group()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 17

`get_name()` (`openscm_units._unit_registry.ScmUnitRegistry`
 method), 17

`get_root_units()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 17

`get_symbol()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 18

`get_system()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 18

I

`is_compatible_with()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 18

L

`load_definitions()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 18

M

`MIXTURES` (in module `openscm_units.data.mixtures`), 23

module

`openscm_units._unit_registry`, 13
`openscm_units.data`, 23

N

`non_int_type` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 attribute), 18

O

`openscm_units._unit_registry`
 module, 13
`openscm_units.data`
 module, 23

P

`parse_expression()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 18
`parse_pattern()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 19
`parse_unit_name()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 19
`parse_units()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 19
`pi_theorem()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 19

R

`remove_context()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 19

S

`ScmUnitRegistry` (class in open-
 `scm_units._unit_registry`), 15
`set_fmt_locale()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 19
`setup_matplotlib()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 20
`split_gas_mixture()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 20
`sys()` (`openscm_units._unit_registry.ScmUnitRegistry`
 property), 20

U

`unit_registry` (in module open-
 `scm_units._unit_registry`), 21

`UnitsContainer()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 15

W

`with_context()` (open-
 `scm_units._unit_registry.ScmUnitRegistry`
 method), 20
`wraps()` (`openscm_units._unit_registry.ScmUnitRegistry`
 method), 20