OpenSCM-Units Documentation

Release 0.1.4+0.gf686799.dirty

Zeb Nicholls, Sven Willner, Jared Lewis, Robert Gieseke

Jan 07, 2021

DOCUMENTATION

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

CHAPTER

ONE

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`.

CHAPTER

THREE

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.

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 intall --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.

CHAPTER

FOUR

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_regsitry` 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(1246.666666666667, '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')>
```

NOx

Like for methane, NOx emissions also suffer from a namespace collision. In order to prevent inadvertent conversions from 'NOx' to e.g. 'N2O', the conversion 'NOx' <–> 'N' is by default forbidden. It can be performed within the 'NOx_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')>
```

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

Unit registry class.

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

add_standards()

Add standard units.

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

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.

openscm_units.unit_registry.**unit_registry = <openscm_units.unit_registry.ScmUnitRegistry ob**
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`.

CHAPTER

FIVE

CHANGELOG

5.1 master

5.2 v0.1.4

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

5.3 v0.1.3

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

5.4 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

5.5 v0.1.1

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

5.6 v0.1.0

- (#1) Setup repository

CHAPTER

SIX

INDEX

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX