OpenSCM Documentation

Release 0+untagged.94.g077f9b5.dirty

Robert Gieseke, Zebedee Nicholls, Sven Willner

Jul 02, 2019

DOCUMENTATION

Warning: OpenSCM is still work in progress and cannot be fully used yet! However, we are very grateful for suggestions and critique on how you would like to use this framework. Please have a look at the issues and feel free to create new ones/upvote ones that would really help you.

The **Open Simple Climate Model framework** unifies access to several simple climate models (SCMs). It defines a standard interface for getting and setting model parameters, input and output data as well as for running the models. Additionally, OpenSCM provides a standardized file format for these parameters and scenarios including functions for reading and writing such files. It further adds convenience functions and easily enables ensemble runs, e.g. for scenario assessment or model tuning.

This OpenSCM implementation comes with a command line tool `openscm`.

CHAPTER

ONE

USE GUIDELINES

We encourage use of OpenSCM as much as possible and are open to collaboration. If you plan to publish using OpenSCM, please be respectful of the work and the *Maintainers*' willingness to open source their code.

In particular, when using OpenSCM, please cite the DOI of the precise version of the package used and consider citing our package description paper [when it's written, which it's not yet :)]. As appropriate, please consider also citing the wrappers and models that OpenSCM relies on. A way to cite OpenSCM alongside the references to the wrappers and original models can be found in the documentation and are available in bibtex format in the `CITATION` file.

Of course, there is a balance, and no single rule will fit all situations. If in doubt, don't hestiate to contact the *Maintainers* and ask.

CHAPTER

TWO

MAINTAINERS

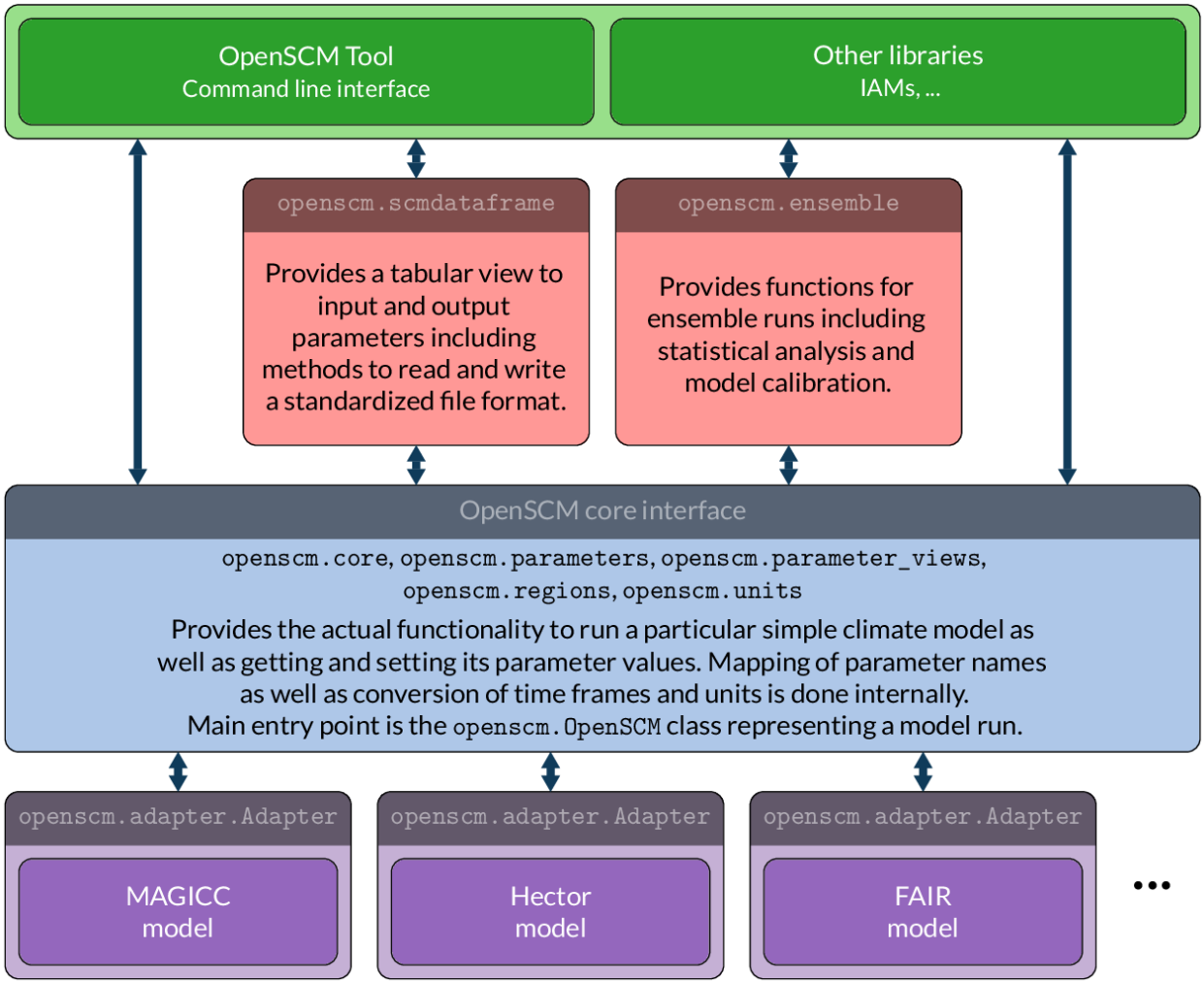
Current maintainers of OpenSCM are:

- Robert Gieseke <robert.gieseke@pik-potsdam.de>
- Jared Lewis <jared.lewis@climate-energy-college.org>
- Zebedee Nicholls <zebedee.nicholls@climate-energy-college.org>
- Sven Willner <sven.willner@pik-potsdam.de>

CHAPTER

THREE

SCHEMA



3.1 Installation

To install OpenSCM run

```
pip install openscm
```

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

```
pip install openscm[notebooks]
```

OpenSCM comes with model adapters only for some very simple SCMs. If you want to run other models, you will also need to install their dependencies (see ReadTheDocs for a list).

3.2 Command line tool

3.3 Quickstart

3.4 Usage

OpenSCM defines and provides three interfaces for its users.

Its **main** interface is targeted to users who want to include OpenSCM in their model, e.g. integrated assessment modellers. This interface provides the basic functionality necessary to run all SCMs included in OpenSCM. It includes functions for getting and setting parameters as well as to run and reset the model. Additionally, it allows for reading and writing parameters from and to standardized and other file formats, including whole scenario definitions.

The **ensemble** interface provides functions for running ensembles of model runs.

The **ScmDataFrame** provides a high-level model output analysis tool. It is designed to make model output filtering, plotting and writing to/from disk as easy as possible. In addition, it integrates with the tools provided by the pyam <https://github.com/IAMconsortium/pyam>_ package which facilitates easy inclusion of data from integrated assessment modelling exercises too.

The **command line** interface lets users run models with specified parameters and model input directly from the command line without coding themselves. Please see *Command line tool* for its usage documentation.

3.4.1 Parameters

Parameter here refers to any named input or output variable of a model. A parameter can be either scalar (i.e. a single number), a timeseries, a boolean, or a string value and has a unique name in a hierarchy of arbitrary depth. That means every parameter either is a root parameter without a parent parameter ("level 0") or belongs to a parent parameter.

For example, the parameter for industrial carbon emissions belongs to the parameter for carbon emissions, which in turn belongs to the root parameter for emissions. Thus, it is identified by

`Emissions` -> `CO2` -> `Industrial`.

In the core API parameters are expected to be identified by tuples of strings describing their position in the hierarchy, i.e. in this example `("Emissions", "CO2", "Industrial")`.

See *Standard parameters* for the standard parameters in OpenSCM.

3.4.2 Time frames

Timeseries parameters are always given with a corresponding time frame, which consists of a time point and a period length. The time point gives the start of the timeseries; the period length gives the length of the period between consecutive values in the timeseries. Each parameter value is assumed to be the **average value** for its corresponding period. This implies that values that are not averages but, for instance, absolute values, such as emissions need to be given as a rate, e.g. `tC/a` rather than `tC`.

3.4.3 Main interface

(see *OpenSCM* for an API reference)

Setting up a model run

A model run is represented by a *openscm.OpenSCM* object specifying the underlying SCM and start and end time:

```
from openscm import OpenSCM

model = OpenSCM("DICE")
```

Setting input parameters

In the core API parameters are get and set through subclasses of `openscm.parameter_views.ParameterView`. While the values of the parameters are stored internaly, a `openscm.parameter_views.ParameterView` provides an (always up-to-date) "view" of the corresponding parameter and will always return the parameter values in a specific unit and, in the case of timeseries, a specific time frame.

Unit and time frame have to be specified when requesting a `ParameterView` from the *OpenSCM*'s `ParameterSet` property called `parameters` using one of the following functions:

- `scalar()` returns a view to a scalar ("number") parameter (`ScalarView`)
- `timeseries_()` returns a view to a timeseries parameter (`TimeseriesView`)
- `generic()` returns a view to a generic parameter, i.e. one of a non-scalar, non-timeseries type, which is not converted in any way (`GenericView`)

Each of these functions take the hierarchical name of the parameter (as described under *Parameters*) and, in a similar fashion, optionally, the hierarchical name of the region it applies to. The "root" region, i.e. the region of which all others are subregions and which applies to parameters for all regions, is by default named `"World"`.

Values can be get and set using the `value` and `values` property for scalar/generic and timeseries views, respectively. Conversion, if necessary, is done internally by the object. There is no standard for the unit and time frame for internal storage, but those of the first `openscm.parameter_views.ParameterView` requested are used. If a scalar view for a time series is requested (or vice-versa), or if the units are not convertible, an error is raised. For timeseries, the conversion also happens after altering (or reading) particular values of the timeseries `values`.

`ParameterView` objects also convert between hierarchical levels if possible: a view to a higher level parameter yields the sum of its child parameters. This implies that, once a view to a parameter has been written to, there cannot be a view to one of its children. Otherwise consistency cannot be guaranteed, so an error is raised. The same holds if a child parameter has already been set and the user tries to set values for one of its parent parameters. A similar logic applies to the hierarchy of regions.

Using `ParameterView` as proxy objects rather than directly setting/returning parameter values allows for efficient parameter handling in the expected units and time frames without specifying these for each value (e.g. seeting a timeseries step-wise would create large overhead).

```
climate_sensitivity = model_run.parameters.scalar(
    "Equilibrium Climate Sensitivity", "degC"
)
climate_sensitivity.value = 3

carbon_emissions_raw = [10 for _ in range(2100 - 2006)]
time_points = create_time_points(
```

(continues on next page)

(continued from previous page)

```
    start_time,
    year_seconds,
    len(carbon_emissions_raw),
    "average",
)
carbon_emissions = model_run.parameters.timeseries(
    ("Emissions", "CO2"),
    "GtCO2/a",
    time_points,
    "average",
)
carbon_emissions.values = carbon_emissions_raw
```

Running the model

The model is simply run by calling the *run()* function:

```
import numpy as np

start_time = np.datetime64("2006-01-01")
stop_time = np.datetime64("2100-01-01")
model.parameter.generic("Start Time").value = start_time
model.parameter.generic("Stop Time").value = stop_time

model.run()
```

This tells the adapter for the particular SCM to get the necessary parameters in the format as expected by the model, while conversion for units and time frames is done by the corresponding `openscm.parameter_views.ParameterView` objects. It then runs the model itself.

After the run the model is reset, so the *run()* function can be called again (setting parameters to new values before, if desired).

Getting output parameters

During the run the model adapter sets the output parameters just like the input parameters were set above. Thus, these can be read using read-only `ParameterView` objects:

```
gmt = model_run.parameters.timeseries(
    ("Surface Temperature", "Increase"), "degC", start_time, year_seconds
)
print(gmt.values)
```

3.5 Models

OpenSCM currently supports the following simple climate models using adapters. See *Writing a model adapter* on how to implement an adapter for a further SCM.

Model name	Name in OpenSCM	URL	Comment
MODELNAME	MODELNAME	awesomemodel.com	

3.5.1 MODELNAME

Describe model here

3.6 Development

```
git clone git@github.com:openclimatedata/openscm.git
pip install -e .
```

Tests can be run locally with

```
python setup.py test
```

3.6.1 Writing a model adapter

Writing adapter tests

To help ensure your adapter works as intended, we provide a number of standard tests. To run these, create a file `test_myadapter.py` in `tests/adapters/` and subclass the `AdapterTester` (this ensures that the standard tests are run on your adapter). Tests are done using pytest on all methods starting with `test_`. Only pull requests with adapters with full test coverage will be merged (see, for instance, the coverage on the end of the PR page).

```
# tests/adapters/test_myadapter.py

from openscm.adapters.myadapter import MyAdapter

from base import _AdapterTester


class TestMyAdapter(_AdapterTester):
    tadapter = MyAdapter

    # if necessary, you can extend the tests e.g.
    def test_run(self, test_adapter, test_run_parameters):
        super().test_run(test_adapter, test_run_parameters)
        # TODO some specific test of your adapter here

    def test_my_special_feature(self, test_adapter):
        # TODO test some special feature of your adapter class
```

Creating an `Adapter` subclass

Create your adapter source file in `openscm/adapters/`, e.g. `myadapter.py`, and subclass the `openscm.adapter.Adapter` class:

```
# openscm/adapters/myadapter.py

from ..adapter import Adapter

YEAR = 365 * 24 * 60 * 60  # example time step length as used below

class MyAdapter(Adapter):
```

Implement the relevant methods (or just do `pass` if you do not need to do anything in the particular method). The only part of OpenSCM with which adapters should interact is `ParameterSet`.

- The `_initialize_model()` method initializes the adapter and is called only once just before the first call to the functions below initializing the first run. It should set the default values of mandatory *model-specific* (not *standard OpenSCM parameters*!) parameters in the in the `ParameterSet` stored in the adapter's `_parameters` attribute. The *hierarchical names* of these model-specific parameters should start with the model/adapter name (as you set it in the model registry, see below).

```
def _initialize_model(self) -> None:
    # TODO Initialize the model
    # TODO Set default parameter values:
    self._parameters.get_writable_scalar_view(
        ("MyModel", "Specific Parameter"), ("World",), "Unit"
    ).set(DEFAULT_VALUE)
```

- The `_initialize_run_parameters()` method initializes a particular run. It is called before the adapter is used in any way and at most once before a call to `_run()` or `_step()`.

```
def _initialize_run_parameters(self) -> None:
    """
    TODO Initialize run parameters by reading model parameters
    from `self._parameters` (see below).
    """
```

  The adapter should later use the start and stop time of the run as stored in the `self._start_time` and `self._stop_time` attributes.

- The `_initialize_model_input()` method initializes the input and model parameters of a particular run. It is also called before the adapter is used in any way and at most once before a call to `_run()` or `_step()`.

  This and the `_initialize_run_parameters()` method are separated for higher efficiency when doing ensemble runs for models that have additional overhead for changing drivers/scenario setup.

```
def _initialize_model_input(self) -> None:
    """
    TODO Initialize model input by reading input parameters from
    :class:`self._parameters
    <~openscm.adapter.Adapter._parameters>` (see below).
    """
```

- The `_reset()` method resets the model to prepare for a new run. It is called once after each call of `_run()` and to reset the model after several calls to `_step()`.

```
def _reset(self) -> None:
    # TODO Reset the model
```

- The `_run()` method runs the model over the full time range (as given by the times set by the previous call to `_initialize_run_parameters()`). You should at least implement this function.

```
def _run(self) -> None:
    """
    TODO Run the model and write output parameters to
    :class:`self._output <~openscm.adapter.Adapter._output>`
    (see below).
    """
```

- The `_step()` method does a single time step. You can get the current time from `self._current_time`, which you should increase by the time step length and return its value. If your model does not support stepping

just do `raise NotImplementedError` here.

```
def _step(self) -> None:
    """
    TODO Do a single time step and write corresponding output
    parameters to :class:`self._output
    <~openscm.adapter.Adapter._output>` (see below).
    """
    self._current_time += YEAR
```

- The `_shutdown()` method cleans up the adapter.

```
def _shutdown(self) -> None:
    # TODO Shut down model
```

Reading model and input parameters and writing output parameters

Model parameters and input data (referred to as general "parameters" in OpenSCM) are **pulled** from the `ParameterSet` provided by the OpenSCM Core. OpenSCM defines a *set of standard parameters* to be shared between different SCMs. As far as possible, adapters should be able to take all of them as input from `_parameters` and should write their values to `_output`.

For efficiency, the OpenSCM Core interface provides subclasses of `ParameterView` that provide a view into a parameter with a requested *time frame* and *unit*. Conversion (aggregation, unit conversion, and time frame adjustment) is done interally if possible. Subclasses implement functionality for scalar and time series values, each for read-only as well as writable views, which you can get from the relevant `ParameterSet` (see *Setting input parameters*).

Accordingly, you should establish the views you need in the `_initialize_model()` method and save them as protected attributes of your adapter class. Then, get their values in the `_initialize_model_input()` and `_initialize_run_parameters()` methods. In the `_run()` and `_step()` methods you should write the relevant output parameters.

Adding the adapter to the model registry

Once done with your implementation, add a lookup for your adapter in `openscm/adapters/__init__.py` (where marked in the file) according to:

```
elif name == "MyAdapter":
    from .myadapter import MyAdapter

    adapter = MyAdapter
```

(make sure to set `adapter` to your *class* not an instance of your adapter)

Additional module dependencies

If your adapter needs additional dependencies add them to the `REQUIREMENTS_MODELS` dictionary in `setup.py` (see comment there).

3.6.2 Contributing

Thanks for contributing to OpenSCM. We are always trying to improve this tool, add new users and can do so even faster with your help!

Following the guidelines will help us work together as efficiently as possible. When we all work with a common understanding, we can make sure that issues are addressed quickly, suggested changes can be easily assessed and pull requests can be finalised painlessly.

All contributions are welcome, some possible suggestions include:

- bug reports (make a new issue and use the template please :D)
- feature requests (make a new issue and use the template please :D)
- pull requests (make a pull request and use the template please :D)
- tutorials (or support questions which, once solved, result in a new tutorial :D)
- improving the documentation

Please don't use the repository to have discussions about the results. Such discussions are scientific and generally belong in the scientific literature, not in a development repository.

Ground Rules

As a contributor, it is vital that we all follow a few conventions:

- Be welcoming to newcomers and encourage diverse new contributors from all backgrounds. See the *Code of Conduct*.
- Create issues for changes and enhancements, this ensures that everyone in the community has a chance to comment
- Ensure that you pass all the tests before making a pull request
- Avoid pushing directly to master, all changes should come via pull requests

Setup

Editor Config

The repository contains a `.editorconfig` file. This ensures that all of our text editors behave the same way and avoids spurious changes simply due to differing whitespace or end of line rules.

Many editors have built in support for `Editorconfig` but some require a plugin. To work out if your editor requires a plugin, please check https://editorconfig.org/.

Getting started

Your First Contribution

The development methodology for OpenSCM makes heavy use of `git`, `make`, virtual environments and test driven development. If you aren't familiar with any of these terms it might be helpful to spend some time getting up to speed on these technologies. Some helpful resources (the longest take about 5 hours to work through):

- Introduction to git by Software Carpentry
- "Making a pull request"
- "My first pull request"
- Intoduction to tests by Software Carpentry and "Getting started with mocking"
- Introduction to make by Software Carpentry

- Virtual environments with venv
- Continuous integration (CI); we use Travis CI for our CI but there are a number of good providers.
- Jupyter Notebooks; we recommend simply installing `jupyter` (`conda install jupyter` or `pip install jupyter`) in your virtual environment.
- Documentation generation with Sphinx

Development workflow

For almost all changes, there should be a corresponding Pull Request (PR) on GitHub to discuss the changes and track the overall implementation of the feature. These PRs should use the PR template.

It is better to break a larger problem into smaller features if you can. Each feature is implemented as a branch and merged into master once all of the tests pass. This is development workflow is preferred to one long-lived branch which can be difficult to merge.

The workflow for implementing a change to opencm is:

- Create a PR. Initially you will not be ready to merge so prefix the title of the PR with 'WIP:'.
- Start a branch for the feature (or bug fix). When you start a new branch, be sure to pull any changes to master first.

```
git checkout master
git pull
git checkout -b my-feature
```

- Develop your feature. Ensure that you run `make test` locally regularly to ensure that the tests still pass
- Push your local development branch. This builds, tests and packages OpenSCM under Linux. The committer will be emailed if this process fails.
- Before the PR can be merged it should be approved by another team member and it must pass the test suite. If you have a particular reviewer in mind, assign the PR to that user.
- Your PR may need to be rebased before it can be merged. Rebasing replays your commits onto the new master commit and allows you to rewrite history.

```
git fetch
git checkout my-feature
git rebase -i origin/master
```

- Once approved, a maintainer can merge the PR.

Testing

The tests are automatically run after every push using GitHub's CI pipelines. If the tests fail, the person who committed the code is alerted via email.

Running the tests

To run the tests locally, simply run `make test`. This will create an isolated virtual environment with the required python libraries. This virtual environment can be manually regenerated using `make venv -B`.

Types of test

We have a number of different types of test:

- unit, in the `tests/unit` folder
- integration, in the `tests/integration` folder

Unit

Unit tests test isolated bits of code, one at a time. Thus, they only work if the tested functions are small and will almost inevitably require the use of mocking. Their purpose is to help to isolate bugs down to particular functions or lines of code.

Integration

Integration tests test a whole pipeline of functions on a higher level than unit tests. They ensure that all our joins make sense when run without (or with few) mocks. Overall, integration tests should reproduce how a user would interact with the package.

Release Process

We use tags to represent released versions of OpenSCM. Once you have tagged a new release in our git respoitory, `versioneer` takes care of the rest.

We follow Semantic Versioning, where version strings are of the format vMAJOR.MINOR.PATCH. We follow these conventions when deciding how to increment the version number, increment

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards-compatible manner
- PATCH version when you make backwards-compatible bug fixes.

The steps undertaken to create a release are:

- Checkout the latest commit in the master branch and ensure that your working copy is clean
- Update `CHANGELOG.rst` to tag the unreleased items with the version and date of release. The unreleased section should now be empty.
- Commit the changes with the message "Bumped to {}" where {} is replaced with the version string
- Tag the commit with the version string. i.e. `git tag v7.1.0`
- Push the commit and tags `git push; git push --tags`

Attribution

Thanks to https://github.com/nayafia/contributing-template/blob/master/CONTRIBUTING-template.md for the template.

3.6.3 Creating a release

OpenSCM uses designated Github Actions to upload the package to PyPI (and, in the future, also to Conda). To create a release:

1. Change the "master" header in `CHANGELOG.rst` to the release version number (not starting with "v", e.g. "1.2.3") and create a new, empty "master" header above. Commit these changes with the message, "Prepare for release of vVERSIONNUMBER" e.g. "Prepare for release of v1.2.3".
2. Tag the commit as "vVERSIONNUMBER", e.g. "v1.2.3", on the "master" branch. Push the tag.
3. The Github Actions workflow should now create a release with the corresponding description in `CHANGELOG.rst` and upload the release to PyPI.

3.6.4 Code of Conduct

We as contributors and maintainers want to foster an open and welcoming environment around the OpenSCM project. To that end, we have a few ground rules that we ask everyone to adhere to. This code applies equally to everyone involved in the project.

This is not an exhaustive code which covers all possible behaviour. Rather, take it in the spirit in which it is intended - a guide to make it easier to enrich all of us and the technical communities in which we participate.

This code of conduct applies to all spaces managed by the OpenSCM project. This includes mailing lists, the issue tracker, group calls, and any other forums of the project. In addition, violations of this code outside these spaces may affect a person's ability to participate within them.

If you believe someone is violating the code of conduct, we ask that you report it by emailing the maintainers listed in the README.

- Be friendly and patient.
- Be welcoming. We strive to be a community that welcomes and supports people of all backgrounds and identities. This includes, but is not limited to members of any race, ethnicity, culture, national origin, color, immigration status, social and economic class, educational level, sex, sexual orientation, gender identity and expression, age, size, family status, political belief, religion, and mental and physical ability.
- Be considerate. Your work will be used by other people, and you in turn will depend on the work of others. Any decision you take will affect users and colleagues, and you should take those consequences into account when making decisions.
- Be respectful. Not all of us will agree all the time, but disagreement is no excuse for poor behavior and poor manners. We might all experience some frustration now and then, but we cannot allow that frustration to turn into a personal attack. It is important to remember that a community where people feel uncomfortable or threatened is not a productive one. Everyone involved in the project should be respectful when dealing with others.
- Be careful in the words that you choose. We want to be a community of professionals, and we conduct ourselves professionally. Be kind to others. Do not insult or put down other participants. Harassment and other exclusionary behavior are not acceptable. This includes, but is not limited to:
  - Violent threats or language directed against another person.
  - Discriminatory jokes and language.
  - Posting sexually explicit or violent material.
  - Posting (or threatening to post) other people's personally identifying information ("doxing").
  - Personal insults, especially those using racist or sexist terms.
  - Unwelcome sexual attention.

  - Advocating for, or encouraging, any of the above behavior.
  - Repeated harassment of others. In general, if someone asks you to stop, then stop.

- When others disagree, try to understand why. Disagreements, both social and technical, happen all the time and this project is no exception. It is important that we resolve disagreements and differing views constructively. Different people have different perspectives on issues. Being unable to understand why someone holds a viewpoint does not mean that they are wrong. Do not forget that it is human to err and blaming each other does not get us anywhere. Instead, focus on helping to resolve issues and learning from mistakes.

This Code of Conduct is adapted from the Django Code of Conduct and the Contributor Covenant. Like these, this document is released under Creative Commons Attribution (CC BY)

3.7 Standard parameters

In OpenSCM a '*parameter*' is any named input or output variable of a model, e.g. CO_2 emissions, equilibrium climate sensitivity, aerosol forcing scaling. As described *here*, parameters are given in a hierarchy, e.g. `Emissions` -> `CO2` -> `Industrial`.

Simple climate models come in many different shapes and forms hence we do not expect them to all be able to do everything. However, to be included in OpenSCM they should make sure their parameters fit into these standard parameters as far as possible to ensure models can be interchanged easily. Of course, model-specific parameters are also able to be used (see also *Writing a model adapter*).

3.7.1 Conventions

In the following, 'pre-industrial' refers to an unperturbed state of the climate. Individual adapters can translate this into whatever year they need to for their model, but they should do such translations with this definition in mind.

'Reference period' refers to the period a given variable is reported relative to the mean of. For example, 'surface temperature relative to a 1961-1990 reference period' refers to surface temperatures relative to the mean of the period 1961-1990. We are not yet sure how best to handle these reference periods in variables, if you have ideas please contribute to the discussions in #167.

3.7.2 Aggregation

Parameters in OpenSCM come as part of a hierarchy, in the following separated by the `|` character. For example, `Emissions|CO2|Energy`. `Emissions|CO2|Energy` is emissions of CO_2 from the energy sub-sector (whatever 'energy' happens to mean in this context). As far as it makes sense, parameters that are higher in the hierarchy (e.g. `Emissions|CO2` is 'higher' than `Emissions|CO2|Energy`) are the sum of all the variables which are one level below them in the hierarchy. For example, if `Emissions|CO2|Energy`, `Emissions|CO2|Transport` and `Emissions|CO2|Agriculture` are provided, `Emissions|CO2` would be the sum of these.

3.7.3 Standards

Standard parameters

Below we provide a list of the OpenSCM standard parameters adapters must adhere to as far as a specific variable concerns them. Alongside we give the type of unit that the parameter should be given in and how it should be expected by adapters. Conversion between particular units is done automatically if possible.

In the following, `<GAS>` can be one of the standard *Gases*.

Table 1: Standard parameters

Parameter name 0	Parameter name 1	Parameter name 2	Unit type	Note
`Start Time`			`np.datetime64` object	Time of the first time step of the model run
`Stop Time`			`np.datetime64` object	Time of the last time step of the model run
`Emissions`	`<GAS>`		mass <GAS> / time	
`Atmospheric Concentrations`	`<GAS>`		parts per X where X is million, billion, trillion etc.	Aggregation possible, but does not always make sense
`Pool`	`<GAS>`		mass <GAS>	
`Radiative Forcing`			power / area	Aggregation gives total forcing, but be carful of double reporting, e.g. providing `Radiative Forcing|Aerosols|Direct Effect` and `Radiative Forcing|Aerosols|NOx`
`Radiative Forcing`	`<GAS>`			
`Radiative Forcing`	`Aerosols`			
`Radiative Forcing`	`Aerosols`	`Direct Effect`		
`Radiative Forcing`	`Aerosols`	`Indirect Effect`		
`Radiative Forcing`	`Aerosols`	`SOx`		
`Radiative Forcing`	`Aerosols`	`NOx`		
`Radiative Forcing`	`Aerosols`	`OC`		
`Radiative Forcing`	`Aerosols`	`BC`		
`Radiative Forcing`	`Land-use Change`			
`Radiative Forcing`	`Black Carbon on Snow`			
`Radiative Forcing`	`Volcanic`			
`Radiative Forcing`	`Solar`			
`Radiative Forcing`	`External`			
`<X> to <Y> Flux`			mass / time	See *Material Fluxes*
`Surface Temperature`			temperature	Surface air temperature i.e. `tas`
`Ocean Temperature`			temperature	Surface ocean temperature i.e. `tos`
`Ocean`			energy	

Gases

Table 2: Gases

Name	Description
CO2	Carbon
CH4	Methane
N2O	Nitrous oxide
SOx	Sulfur oxide
CO	Carbon monoxide
NMVOC	Volatile organic compound
NOx	Nitrogen oxide
BC	Black carbon
OC	Organic carbon
NH3	NH3
NF3	NF3
CF4	CF4
C2F6	C2F6
C3F8	C3F8
cC4F8	cC4F8
C4F10	C4F10
C5F12	C5F12
C6F14	C6F14
C7F16	C7F16
C8F18	C8F18
CCl4	CCl4
CHCl3	CHCl3
CH2Cl2	CH2Cl2
CH3CCl3	CH3CCl3
CH3Cl	CH3Cl
CH3Br	CH3Br
HFC23	HFC23
HFC32	HFC32
HFC4310	HFC4310
HFC125	HFC125
HFC134a	HFC134a
HFC143a	HFC143a
HFC152a	HFC152a
HFC227ea	HFC227ea
HFC236fa	HFC236fa
HFC245fa	HFC245fa
HFC365mfc	HFC365mfc
CFC11	CFC11
CFC12	CFC12
CFC113	CFC113
CFC114	CFC114
CFC115	CFC115
HCFC22	HCFC22
HCFC141b	HCFC141b
HCFC142b	HCFC142b
SF6	SF6

Continued on next page

Table 2 – continued from previous page

Name	Description
SO2F2	SO2F2
Halon1202	Halon1202
Halon1211	Halon1211
Halon1301	Halon1301
Halon2402	Halon2402

Material Fluxes

These variables can be used to store the flux of material within the model. They should be of the form <X> to <Y> Flux where the material is flowing from <X> into <Y> (and hence negative values represent flows from <Y> into <X>):

- Land to Air Flux|CO2|Permafrost (mass carbon / time) - land to air flux of CO_2 from permafrost
- Land to Air Flux|CH4|Permafrost (mass methane / time)

3.7.4 Standard regions

Similarly to variables, regions are also given in a hierarchy. Regions which are higher in the hierarchy are the sum of all the regions which are one level below them in the hierarchy (be careful of this when looking at e.g. CO_2 concentration data at a regional level).

The hemispheric regions should be fairly obvious and well-defined. The land/ocean split is somewhat fuzzily defined as the transition between land and ocean does not have a precise definition. We don't provide a clear definition because a) there isn't an agreed one in the literature and b) no simple climate model is detailed enough for the slight fuzziness around these definitions to matter. We choose to put the hemispheres before the ocean/land split in the hierarchy because it makes more sense to us but are happy to discuss further if desired (raise an issue).

Descriptions of the rest of the regions can be found in the 'Description' column below.

Warning Be careful, if you mix multiple regional conventions (e.g. reporting both ("World", "Land") and ("World", "R5ASIA")), then your ("World") total will double count some quantities and so may provide misleading information. There is no way for OpenSCM to reasonably keep track of what overlaps with what so we can't automate this process (if you think you have an idea of how to do this, please make a PR :D).

Table 3: Gases

Name 0	Name 1	Name 2	Description
World			Entire globe
World	Northern Hemisphere		Northern hemisphere
World	Northern Hemisphere	Ocean	Northern hemisphere ocean
World	Northern Hemisphere	Land	Northern hemisphere land
World	Southern Hemisphere		Southern hemisphere
World	Southern Hemisphere	Ocean	Southern hemisphere ocean
World	Southern Hemisphere	Land	Southern hemisphere land
World	Ocean		Ocean
World	Land		Land
World	R5ASIA		Non-OECD Asia - see IIASA AR5 database
World	R5REF		Reforming economies of Eastern Europe and the Former Soviet Union (also known as R5EIT i.e. economies in transition) - see IIASA AR5 database
World	R5MAF		Middle East and Africa - see IIASA AR5 database
World	R5OECD		OECD - see IIASA AR5 database
World	R5LAM		Latin America and the Caribbean - see IIASA AR5 database
World	R5.2ASIA		Most Asian countries - see IIASA SSP database
World	R5.2REF		Reforming economies of Eastern Europe and the Former Soviet Union - see IIASA SSP database
World	R5.2MAF		Middle East and Africa - see IIASA SSP database
World	R5.2OECD		OECD - see IIASA SSP database
World	R5.2LAM		Latin America and the Caribbean - see IIASA SSP database
World	Bunkers		Typically used to capture all non-country associated emissions i.e. international shipping (and sometimes aviation) - be careful with definition

3.8 File formats

3.9 OpenSCM

class opensmc.**OpenSCM**(*model_name*, *input_parameters=None*, *output_parameters=None*)

Bases: object

OpenSCM core class.

Represents a particular simple climate model to be run.

__init__(*model_name*, *input_parameters=None*, *output_parameters=None*)

Initialize.

Parameters

- **model** – Name of the SCM to run
- **input_parameters** (Optional[ParameterSet]) – Input ParameterSet to use (or a new one is used when this is None)

- **output_parameters** (Optional[ParameterSet]) – Output ParameterSet to use (or a new one is used when this is None)

Raises **KeyError** – No adapter for SCM named *model* found

_model = None
Adapter of the SCM to run

_model_name = None
Name of the SCM to run

_output = None
Set of output *parameters* of the model

_parameters = None
Set of input *parameters* for the model

property model
Name of the SCM

Return type str

property output
Set of output parameters of the model

Return type ParameterSet

property parameters
Set of input parameters for the model

Return type ParameterSet

reset_stepping()
Reset the model before starting stepping.

Return type None

run()
Run the model over the full time range.

Return type None

step()
Do a single time step.

Returns Current time

Return type int

3.10 Errors

Errors/Exceptions defined and used in OpenSCM.

exception openscm.errors.**AdapterNeedsModuleError**
Bases: Exception

Exception raised when an adapter needs a module that is not installed.

__init__()
Initialize self. See help(type(self)) for accurate signature.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openscm.errors.**InsufficientDataError**
Bases: ValueError

Exception raised when not enough data is available to convert from one timeseries to another (e.g. when the target timeseries is outside the range of the source timeseries) or when data is too short (fewer than 3 data points).

__init__()
Initialize self. See help(type(self)) for accurate signature.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openscm.errors.**ParameterAggregationError**
Bases: *openscm.errors.ParameterError*

Exception raised when a parameter is read from but has child parameters which cannot be aggregated (boolean or string parameters).

__init__()
Initialize self. See help(type(self)) for accurate signature.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openscm.errors.**ParameterEmptyError**
Bases: *openscm.errors.ParameterError*

Exception raised when trying to read when a parameter's value hasn't been set

__init__()
Initialize self. See help(type(self)) for accurate signature.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openscm.errors.**ParameterError**
Bases: Exception

Exception relating to a parameter. Used as super class.

__init__()
Initialize self. See help(type(self)) for accurate signature.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openscm.errors.**ParameterReadError**
Bases: *openscm.errors.ParameterError*

Exception raised when a parameter has been read from (raised, e.g., when attempting to create a child parameter).

__init__()
Initialize self. See help(type(self)) for accurate signature.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openscm.errors.**ParameterReadonlyError**
Bases: *openscm.errors.ParameterError*

Exception raised when a requested parameter is read-only.

This can happen, for instance, if a parameter's parent parameter in the parameter hierarchy has already been requested as writable.

__init__()
Initialize self. See help(type(self)) for accurate signature.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openscm.errors.**ParameterTypeError**
Bases: *openscm.errors.ParameterError*

Exception raised when a parameter is of a different type than requested.

__init__()
Initialize self. See help(type(self)) for accurate signature.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openscm.errors.**ParameterWrittenError**
Bases: *openscm.errors.ParameterError*

Exception raised when a parameter has been written to (raised, e.g., when attempting to create a child parameter).

__init__()
Initialize self. See help(type(self)) for accurate signature.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openscm.errors.**RegionAggregatedError**
Bases: Exception

Exception raised when a region has already been read from in a region-aggregated way.

__init__()
Initialize self. See help(type(self)) for accurate signature.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception openscm.errors.**TimeseriesPointsValuesMismatchError**
Bases: IndexError

Exception raised when the length of the values and of the time points of a timeseries mismatch (depending on the type of timeseries these must equal or deviate by one).

__init__()
Initialize self. See help(type(self)) for accurate signature.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

3.11 Scenarios

Scenarios included as part of OpenSCM.

These are limited to a select few scenarios which are widely used. They are included because they provide a useful set of scenarios with which we can run tests, make example notebooks and allow users to easily get started.

The RCP data originally came from PIK and has since been re-written into a format which can be read by OpenSCM using the pymagicc package. We are not currently planning on importing Pymagicc's readers into OpenSCM by default, please raise an issue here if you would like us to consider doing so.

openscm.scenarios.**_here = '/home/docs/checkouts/readthedocs.org/user_builds/openscm/envs/la**
RCP emissions data

Type *ScmDataFrame*

3.12 ScmDataFrame

3.12.1 ScmDataFrame

ScmDataFrame provides a high level analysis tool for simple climate model relevant data. It provides a simple interface for reading/writing, subsetting and visualising model data. ScmDataFrames are able to hold multiple model runs which aids in analysis of ensembles of model runs.

class openscm.scmdataframe.**ScmDataFrame**(*data*, *index=None*, *columns=None*, ***kwargs*)
Bases: *openscm.scmdataframe.base.ScmDataFrameBase*

OpenSCM's custom DataFrame implementation.

The ScmDataFrame implements a subset of the functionality provided by pyam's IamDataFrame, but is focused on providing a performant way of storing time series data and the metadata associated with those time series.

For users who wish to take advantage of all of Pyam's functionality, please cast your ScmDataFrame to an IamDataFrame first with *to_iamdataframe()*. Note: this operation can be computationally expensive for large data sets because IamDataFrames stored data in long/tidy form internally rather than ScmDataFrames' more compact internal format.

__init__(*data*, *index=None*, *columns=None*, ***kwargs*)
Initialize.

Parameters

- **data** (Union[*ScmDataFrameBase*, None, DataFrame, Series, ndarray, str]) – A pd.DataFrame or data file with IAMC-format data columns, or a numpy array of timeseries data if `columns` is specified. If a string is passed, data will be attempted to be read from file.
- **index** (Optional[Any]) – Only used if `columns` is not `None`. If `index` is not `None`, too, then this value sets the time index of the `ScmDataFrameBase` instance. If `index` is `None` and `columns` is not `None`, the index is taken from `data`.
- **columns** (Optional[Dict[str, list]]) – If None, ScmDataFrameBase will attempt to infer the values from the source. Otherwise, use this dict to write the metadata for each timeseries in data. For each metadata key (e.g. "model", "scenario"), an array of values (one per time series) is expected. Alternatively, providing a list of length 1 applies the same value to all timeseries in data. For example, if you had three timeseries from 'rcp26' for 3 different models 'model', 'model2' and 'model3', the column dict would look like either 'col_1' or 'col_2':

```
>>> col_1 = {
    "scenario": ["rcp26"],
    "model": ["model1", "model2", "model3"],
    "region": ["unspecified"],
    "variable": ["unspecified"],
    "unit": ["unspecified"]
}
>>> col_2 = {
    "scenario": ["rcp26", "rcp26", "rcp26"],
    "model": ["model1", "model2", "model3"],
    "region": ["unspecified"],
    "variable": ["unspecified"],
    "unit": ["unspecified"]
}
>>> assert pd.testing.assert_frame_equal(
    ScmDataFrameBase(d, columns=col_1).meta,
    ScmDataFrameBase(d, columns=col_2).meta
)
```

- ****kwargs** – Additional parameters passed to `pyam.core._read_file()` to read files

Raises

- **ValueError** – If metadata for ['model', 'scenario', 'region', 'variable', 'unit'] is not found. A `ValueError` is also raised if you try to load from multiple files at once. If you wish to do this, please use `df_append()` instead.
- **TypeError** – Timeseries cannot be read from `data`

_apply_filters(*filters*, *has_nan=True*)

Determine rows to keep in data for given set of filters.

Parameters

- **filters** (Dict[~KT, ~VT]) – Dictionary of filters (`{col: values}`); uses a pseudo-regexp syntax by default but if `filters["regexp"]` is `True`, regexp is used directly.
- **has_nan** (bool) – If *True*', convert all nan values in `meta_col` to empty string before applying filters. This means that "" and "*" will match rows with `np.nan`. If `False`, the conversion is not applied and so a search in a string column which contains `np.nan` will result in a `TypeError`.

Returns Two boolean `np.ndarray`'s. The first contains the columns to keep (i.e. which time points to keep). The second contains the rows to keep (i.e. which metadata matched the filters).

Return type `np.ndarray` of `bool`, `np.ndarray` of `bool`

Raises **`ValueError`** – Filtering cannot be performed on requested column

`_day_match`(*values*)

`_sort_meta_cols`()

`append`(*other*, *inplace=False*, *duplicate_msg='warn'*, ***kwargs*)

Append additional data to the current dataframe.

For details, see `df_append()`.

Parameters

- **`other`** (`Union`[*`ScmDataFrameBase`*, `None`, `DataFrame`, `Series`, `ndarray`, `str`]) – Data (in format which can be cast to `ScmDataFrameBase`) to append
- **`inplace`** (`bool`) – If `True`, append data in place and return `None`. Otherwise, return a new `ScmDataFrameBase` instance with the appended data.
- **`duplicate_msg`** (`Union`[`str`, `bool`]) – If "warn", raise a warning if duplicate data is detected. If "return", return the joint dataframe (including duplicate timeseries) so the user can inspect further. If `False`, take the average and do not raise a warning.
- **`**kwargs`** – Keywords to pass to `ScmDataFrameBase.__init__()` when reading `other`

Returns If not `inplace`, return a new `ScmDataFrameBase` instance containing the result of the append.

Return type `ScmDataFrameBase`

`convert_unit`(*unit*, *context=None*, *inplace=False*, ***kwargs*)

Convert the units of a selection of timeseries.

Uses `openscm.units.UnitConverter` to perform the conversion.

Parameters

- **`unit`** (`str`) – Unit to convert to. This must be recognised by `UnitConverter`.
- **`context`** (`Optional`[`str`]) – Context to use for the conversion i.e. which metric to apply when performing CO2-equivalent calculations. If `None`, no metric will be applied and CO2-equivalent calculations will raise `DimensionalityError`.
- **`inplace`** (`bool`) – If `True`, the operation is performed inplace, updating the underlying data. Otherwise a new `ScmDataFrameBase` instance is returned.
- **`**kwargs`** – Extra arguments which are passed to `filter()` to limit the timeseries which are attempted to be converted. Defaults to selecting the entire ScmDataFrame, which will likely fail.

Returns If `inplace` is not `False`, a new `ScmDataFrameBase` instance with the converted units.

Return type `ScmDataFrameBase`

`copy`()

Return a `copy.deepcopy()` of self.

Returns `copy.deepcopy()` of `self`

Return type ScmDataFrameBase

data_hierarchy_separator = '|'

filter(*keep=True*, *inplace=False*, *has_nan=True*, ***kwargs*)
Return a filtered ScmDataFrame (i.e., a subset of the data).

Parameters

- **keep** (bool) – If True, keep all timeseries satisfying the filters, otherwise drop all the timeseries satisfying the filters
- **inplace** (bool) – If True, do operation inplace and return None
- **has_nan** (bool) – If True, convert all nan values in meta_col to empty string before applying filters. This means that "" and "*" will match rows with np.nan. If False, the conversion is not applied and so a search in a string column which contains ;class:*np.nan* will result in a TypeError.
- ****kwargs** – Argument names are keys with which to filter, values are used to do the filtering. Filtering can be done on:
  - all metadata columns with strings, "*" can be used as a wildcard in search strings
  - 'level': the maximum "depth" of IAM variables (number of hierarchy levels, excluding the strings given in the 'variable' argument)
  - 'time': takes a datetime.datetime or list of datetime.datetime's TODO: default to np.datetime64
  - 'year', 'month', 'day', hour': takes an int or list of int's ('month' and 'day' also accept str or list of str)

  If regexp=True is included in kwargs then the pseudo-regexp syntax in pattern_match is disabled.

Returns If not inplace, return a new instance with the filtered data.

Return type ScmDataFrameBase

Raises **AssertionError** – Data and meta become unaligned

head(**args*, ***kwargs*)
Return head of self.timeseries().

Parameters

- ***args** – Passed to self.timeseries().head()
- ****kwargs** – Passed to self.timeseries().head()

Returns Tail of self.timeseries()

Return type pd.DataFrame

interpolate(*target_times*, *interpolation_type=<InterpolationType.LINEAR: 1>*, *extrapolation_type=<ExtrapolationType.CONSTANT: 0>*)
Interpolate the dataframe onto a new time frame.

Uses openscm.timeseries_converter.TimeseriesConverter internally. For each time series a ParameterType is guessed from the variable name. To override the guessed parameter type, specify a "parameter_type" meta column before calling interpolate. The guessed parameter types are returned in meta.

Parameters

- **target_times** (Union[ndarray, List[Union[datetime, int]]]) – Time grid onto which to interpolate
- **interpolation_type** (Union[*InterpolationType*, str]) – How to interpolate the data between timepoints
- **extrapolation_type** (Union[*ExtrapolationType*, str]) – If and how to extrapolate the data beyond the data in `self.timeseries()`

Returns A new `ScmDataFrameBase` containing the data interpolated onto the `target_times` grid

Return type `ScmDataFrameBase`

line_plot(*x='time'*, *y='value'*, ***kwargs*)
Plot a line chart.

See `pyam.IamDataFrame.line_plot()` for more information.

Return type `None`

property meta
Metadata

Return type `DataFrame`

pivot_table(*index*, *columns*, ***kwargs*)
Pivot the underlying data series.

See `pyam.core.IamDataFrame.pivot_table()` for details.

Return type `DataFrame`

process_over(*cols*, *operation*, ***kwargs*)
Process the data over the input columns.

Parameters

- **cols** (Union[str, List[str]]) – Columns to perform the operation on. The timeseries will be grouped by all other columns in *meta*.
- **operation** (*['median', 'mean', 'quantile']*) – The operation to perform. This uses the equivalent pandas function. Note that quantile means the value of the data at a given point in the cumulative distribution of values at each point in the timeseries, for each timeseries once the groupby is applied. As a result, using `q=0.5` is is the same as taking the median and not the same as taking the mean/average.
- ****kwargs** – Keyword arguments to pass to the pandas operation

Returns The quantiles of the timeseries, grouped by all columns in *meta* other than `cols`

Return type `pd.DataFrame`

Raises **ValueError** – If the operation is not one of [‘median’, ‘mean’, ‘quantile’]

region_plot(***kwargs*)
Plot regional data for a single model, scenario, variable, and year.

See `pyam.plotting.region_plot` for details.

Return type `None`

relative_to_ref_period_mean(*append_str=None*, ***kwargs*)
Return the timeseries relative to a given reference period mean.

The reference period mean is subtracted from all values in the input timeseries.

Parameters

- **append_str** (Optional[str]) – String to append to the name of all the variables in the resulting DataFrame to indicate that they are relevant to a given reference period. E.g. *'rel. to 1961-1990'*. If None, this will be autofilled with the keys and ranges of kwargs.
- ****kwargs** – Arguments to pass to *filter()* to determine the data to be included in the reference time period. See the docs of *filter()* for valid options.

Returns DataFrame containing the timeseries, adjusted to the reference period mean

Return type pd.DataFrame

rename(*mapping*, *inplace=False*)

Rename and aggregate column entries using groupby.sum() on values. When renaming models or scenarios, the uniqueness of the index must be maintained, and the function will raise an error otherwise.

Parameters

- **mapping** (Dict[str, Dict[str, str]]) – For each column where entries should be renamed, provide current name and target name

```
{<column name>: {<current_name_1>: <target_name_1>,
                 <current_name_2>: <target_name_2>}}
```

- **inplace** (bool) – If True, do operation inplace and return None

Returns If inplace is True, return a new ScmDataFrameBase instance

Return type ScmDataFrameBase

Raises **ValueError** – Column is not in meta or renaming will cause non-unique metadata

resample(*rule='AS'*, ***kwargs*)

Resample the time index of the timeseries data onto a custom grid.

This helper function allows for values to be easily interpolated onto annual or monthly timesteps using the rules='AS' or 'MS' respectively. Internally, the interpolate function performs the regridding.

Parameters

- **rule** (str) – See the pandas user guide for a list of options. Note that Business-related offsets such as "BusinessDay" are not supported.
- ****kwargs** – Other arguments to pass through to *interpolate()*

Returns New ScmDataFrameBase instance on a new time index

Return type ScmDataFrameBase

Examples

Resample a dataframe to annual values

```
>>> scm_df = ScmDataFrame(
...     pd.Series([1, 2, 10], index=(2000, 2001, 2009)),
...     columns={
...         "model": ["a_iam"],
...         "scenario": ["a_scenario"],
...         "region": ["World"],
...         "variable": ["Primary Energy"],
...         "unit": ["EJ/y"],
```

(continues on next page)

(continued from previous page)

```
...     }
... )
>>> scm_df.timeseries().T
model             a_iam
scenario     a_scenario
region            World
variable Primary Energy
unit               EJ/y
year
2000                  1
2010                 10
```

An annual timeseries can be the created by interpolating to the start of years using the rule 'AS'.

```
>>> res = scm_df.resample('AS')
>>> res.timeseries().T
model                         a_iam
scenario                 a_scenario
region                       World
variable            Primary Energy
unit                         EJ/y
time
2000-01-01 00:00:00       1.000000
2001-01-01 00:00:00       2.001825
2002-01-01 00:00:00       3.000912
2003-01-01 00:00:00       4.000000
2004-01-01 00:00:00       4.999088
2005-01-01 00:00:00       6.000912
2006-01-01 00:00:00       7.000000
2007-01-01 00:00:00       7.999088
2008-01-01 00:00:00       8.998175
2009-01-01 00:00:00      10.00000
```

```
>>> m_df = scm_df.resample('MS')
>>> m_df.timeseries().T
model                         a_iam
scenario                 a_scenario
region                       World
variable            Primary Energy
unit                         EJ/y
time
2000-01-01 00:00:00       1.000000
2000-02-01 00:00:00       1.084854
2000-03-01 00:00:00       1.164234
2000-04-01 00:00:00       1.249088
2000-05-01 00:00:00       1.331204
2000-06-01 00:00:00       1.416058
2000-07-01 00:00:00       1.498175
2000-08-01 00:00:00       1.583029
2000-09-01 00:00:00       1.667883
                             ...
2008-05-01 00:00:00       9.329380
2008-06-01 00:00:00       9.414234
2008-07-01 00:00:00       9.496350
2008-08-01 00:00:00       9.581204
2008-09-01 00:00:00       9.666058
```

(continues on next page)

(continued from previous page)

```
2008-10-01 00:00:00       9.748175
2008-11-01 00:00:00       9.833029
2008-12-01 00:00:00       9.915146
2009-01-01 00:00:00      10.000000
[109 rows x 1 columns]
```

Note that the values do not fall exactly on integer values as not all years are exactly the same length.

References

See the pandas documentation for *resample <http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.resample.html>* for more information about possible arguments.

scatter(*x*, *y*, ***kwargs*)
Plot a scatter chart using metadata columns.

See `pyam.plotting.scatter()` for details.

Return type `None`

set_meta(*meta*, *name=None*, *index=None*)
Set metadata information.

TODO: re-write this to make it more sane and add type annotations

Parameters

- **meta** (`Union[Series, list, int, float, str]`) – Column to be added to metadata
- **name** (`Optional[str]`) – Meta column name (defaults to `meta.name`)
- **index** (`Union[DataFrame, Series, Index, MultiIndex, None]`) – The index to which the metadata is to be applied

Raises **ValueError** – No name can be determined from inputs or index cannot be coerced to `pd.MultiIndex`

Return type `None`

tail(**args*, ***kwargs*)
Return tail of `self.timeseries()`.

Parameters

- ***args** – Passed to `self.timeseries().tail()`
- ****kwargs** – Passed to `self.timeseries().tail()`

Returns Tail of `self.timeseries()`

Return type `pd.DataFrame`

property time_points
Time points of the data

Return type `ndarray`

timeseries(*meta=None*)
Return the data in wide format (same as the timeseries method of `pyam.IamDataFrame`).

Parameters **meta** (`Optional[List[str]]`) – The list of meta columns that will be included in the output's MultiIndex. If None (default), then all metadata will be used.

Returns DataFrame with datetimes as columns and timeseries as rows. Metadata is in the index.

Return type pd.DataFrame

Raises **ValueError** – If the metadata are not unique between timeseries

to_csv(*path*, ***kwargs*)
Write timeseries data to a csv file

Parameters **path** (str) – Path to write the file into

Return type None

to_iamdataframe()
Convert to a LongDatetimeIamDataFrame instance.

LongDatetimeIamDataFrame is a subclass of pyam.IamDataFrame. We use LongDatetimeIamDataFrame to ensure all times can be handled, see docstring of LongDatetimeIamDataFrame for details.

Returns LongDatetimeIamDataFrame instance containing the same data.

Return type LongDatetimeIamDataFrame

Raises **ImportError** – If pyam is not installed

to_parameterset(*parameterset=None*)
Add parameters in this ScmDataFrameBase to a ParameterSet.

It can only be transformed if all timeseries have the same metadata. This is typically the case if all data comes from a single scenario/model input dataset. If that is not the case, further filtering is needed to reduce to a dataframe with identical metadata.

Parameters **parameterset** (Optional[ParameterSet]) – ParameterSet to add this ScmDataFrameBase's parameters to. A new ParameterSet is created if this is None.

Returns ParameterSet containing the data in self (equals parameterset if not None)

Return type ParameterSet

Raises **ValueError** – Not all timeseries have the same metadata or climate_model is given and does not equal "unspecified"

property values
Timeseries values without metadata

Calls *timeseries()*

Return type ndarray

openscm.scmdataframe.**convert_openscm_to_scmdataframe**(*parameterset*, *time_points*, *model='unspecified'*, *scenario='unspecified'*, *climate_model='unspecified'*)
Get an *ScmDataFrame* from a ParameterSet.

An ScmDataFrame is a view with a common time index for all time series. All metadata in the ParameterSet must be represented as Generic parameters with in the *World* region.

TODO: overhaul this function and move to an appropriate location

Parameters

- **parameterset** (ParameterSet) – ParameterSet containing time series and optional metadata.
- **time_points** (ndarray) – Time points onto which all timeseries will be interpolated.

- **model** (str) – Default value for the model metadata value. This value is only used if the model parameter is not found.
- **scenario** (str) – Default value for the scenario metadata value. This value is only used if the scenario parameter is not found.
- **climate_model** (str) – Default value for the climate_model metadata value. This value is only used if the climate_model parameter is not found.

Raises **ValueError** – If a generic parameter cannot be mapped to an ScmDataFrame meta table. This happens if the parameter has a region which is not `('World',)`.

Returns *ScmDataFrame* containing the data from parameterset

Return type *ScmDataFrame*

3.12.2 Base

Base and utilities for OpenSCM's custom DataFrame implementation.

openscm.scmdataframe.base.**REQUIRED_COLS = ['model', 'scenario', 'region', 'variable', 'unit**
Minimum metadata columns required by an ScmDataFrame

class openscm.scmdataframe.base.**ScmDataFrameBase**(*data*, *index=None*, *columns=None*, ***kwargs*)

Bases: object

Base of OpenSCM's custom DataFrame implementation.

This base is the class other libraries can subclass. Having such a subclass avoids a potential circularity where e.g. OpenSCM imports ScmDataFrame as well as Pymagicc, but Pymagicc wants to import ScmDataFrame too. Hence, importing ScmDataFrame requires importing ScmDataFrame, causing a circularity.

__init__(*data*, *index=None*, *columns=None*, ***kwargs*)
Initialize.

Parameters

- **data** (Union[*ScmDataFrameBase*, None, DataFrame, Series, ndarray, str]) – A pd.DataFrame or data file with IAMC-format data columns, or a numpy array of timeseries data if columns is specified. If a string is passed, data will be attempted to be read from file.
- **index** (Optional[Any]) – Only used if columns is not None. If index is not None, too, then this value sets the time index of the *ScmDataFrameBase* instance. If index is None and columns is not None, the index is taken from data.
- **columns** (Optional[Dict[str, list]]) – If None, ScmDataFrameBase will attempt to infer the values from the source. Otherwise, use this dict to write the metadata for each timeseries in data. For each metadata key (e.g. "model", "scenario"), an array of values (one per time series) is expected. Alternatively, providing a list of length 1 applies the same value to all timeseries in data. For example, if you had three timeseries from 'rcp26' for 3 different models 'model', 'model2' and 'model3', the column dict would look like either 'col_1' or 'col_2':

```
>>> col_1 = {
    "scenario": ["rcp26"],
    "model": ["model1", "model2", "model3"],
    "region": ["unspecified"],
    "variable": ["unspecified"],
```

(continues on next page)

(continued from previous page)

```
    "unit": ["unspecified"]
}
>>> col_2 = {
    "scenario": ["rcp26", "rcp26", "rcp26"],
    "model": ["model1", "model2", "model3"],
    "region": ["unspecified"],
    "variable": ["unspecified"],
    "unit": ["unspecified"]
}
>>> assert pd.testing.assert_frame_equal(
    ScmDataFrameBase(d, columns=col_1).meta,
    ScmDataFrameBase(d, columns=col_2).meta
)
```

- ****kwargs** – Additional parameters passed to `pyam.core._read_file()` to read files

Raises

- **ValueError** – If metadata for ['model', 'scenario', 'region', 'variable', 'unit'] is not found. A `ValueError` is also raised if you try to load from multiple files at once. If you wish to do this, please use *df_append()* instead.
- **TypeError** – Timeseries cannot be read from `data`

_apply_filters(*filters*, *has_nan=True*)

Determine rows to keep in data for given set of filters.

Parameters

- **filters** (`Dict`[~KT, ~VT]) – Dictionary of filters (`{col: values}`); uses a pseudo-regexp syntax by default but if `filters["regexp"]` is `True`, regexp is used directly.
- **has_nan** (`bool`) – If *True'*, convert all nan values in `meta_col` to empty string before applying filters. This means that "" and "*" will match rows with `np.nan`. If `False`, the conversion is not applied and so a search in a string column which contains `np.nan` will result in a `TypeError`.

Returns Two boolean `np.ndarray`'s. The first contains the columns to keep (i.e. which time points to keep). The second contains the rows to keep (i.e. which metadata matched the filters).

Return type `np.ndarray` of `bool`, `np.ndarray` of `bool`

Raises **ValueError** – Filtering cannot be performed on requested column

_data = None

Timeseries data

_day_match(*values*)

_meta = None

Meta data

_sort_meta_cols()

_time_points = None

Time points

append(*other*, *inplace=False*, *duplicate_msg='warn'*, ***kwargs*)

Append additional data to the current dataframe.

For details, see *df_append()*.

Parameters

- **other** (Union[*ScmDataFrameBase*, None, DataFrame, Series, ndarray, str]) – Data (in format which can be cast to *ScmDataFrameBase*) to append
- **inplace** (bool) – If True, append data in place and return None. Otherwise, return a new *ScmDataFrameBase* instance with the appended data.
- **duplicate_msg** (Union[str, bool]) – If "warn", raise a warning if duplicate data is detected. If "return", return the joint dataframe (including duplicate timeseries) so the user can inspect further. If False, take the average and do not raise a warning.
- ****kwargs** – Keywords to pass to *ScmDataFrameBase.__init__()* when reading other

Returns If not inplace, return a new *ScmDataFrameBase* instance containing the result of the append.

Return type *ScmDataFrameBase*

convert_unit(*unit*, *context=None*, *inplace=False*, ***kwargs*)

Convert the units of a selection of timeseries.

Uses openscm.units.UnitConverter to perform the conversion.

Parameters

- **unit** (str) – Unit to convert to. This must be recognised by UnitConverter.
- **context** (Optional[str]) – Context to use for the conversion i.e. which metric to apply when performing CO2-equivalent calculations. If None, no metric will be applied and CO2-equivalent calculations will raise DimensionalityError.
- **inplace** (bool) – If True, the operation is performed inplace, updating the underlying data. Otherwise a new *ScmDataFrameBase* instance is returned.
- ****kwargs** – Extra arguments which are passed to *filter()* to limit the timeseries which are attempted to be converted. Defaults to selecting the entire ScmDataFrame, which will likely fail.

Returns If inplace is not False, a new *ScmDataFrameBase* instance with the converted units.

Return type *ScmDataFrameBase*

copy()

Return a copy.deepcopy() of self.

Returns copy.deepcopy() of self

Return type *ScmDataFrameBase*

data_hierarchy_separator = '|'

String used to define different levels in our data hierarchies.

By default we follow pyam and use "|". In such a case, emissions of CO2 for energy from coal would be "Emissions|CO2|Energy|Coal".

Type str

filter(*keep=True*, *inplace=False*, *has_nan=True*, ***kwargs*)

Return a filtered ScmDataFrame (i.e., a subset of the data).

Parameters

- **keep** (bool) – If True, keep all timeseries satisfying the filters, otherwise drop all the timeseries satisfying the filters
- **inplace** (bool) – If True, do operation inplace and return None
- **has_nan** (bool) – If True, convert all nan values in meta_col to empty string before applying filters. This means that "" and "*" will match rows with np.nan. If False, the conversion is not applied and so a search in a string column which contains ;class:*np.nan* will result in a TypeError.
- ****kwargs** – Argument names are keys with which to filter, values are used to do the filtering. Filtering can be done on:
  - all metadata columns with strings, "*" can be used as a wildcard in search strings
  - 'level': the maximum "depth" of IAM variables (number of hierarchy levels, excluding the strings given in the 'variable' argument)
  - 'time': takes a datetime.datetime or list of datetime.datetime's TODO: default to np.datetime64
  - 'year', 'month', 'day', hour': takes an int or list of int's ('month' and 'day' also accept str or list of str)

  If regexp=True is included in kwargs then the pseudo-regexp syntax in pattern_match is disabled.

Returns If not inplace, return a new instance with the filtered data.

Return type *ScmDataFrameBase*

Raises **AssertionError** – Data and meta become unaligned

head(**args*, ***kwargs*)

Return head of self.timeseries().

Parameters

- ***args** – Passed to self.timeseries().head()
- ****kwargs** – Passed to self.timeseries().head()

Returns Tail of self.timeseries()

Return type pd.DataFrame

interpolate(*target_times*, *interpolation_type=<InterpolationType.LINEAR: 1>*, *extrapolation_type=<ExtrapolationType.CONSTANT: 0>*)

Interpolate the dataframe onto a new time frame.

Uses openscm.timeseries_converter.TimeseriesConverter internally. For each time series a ParameterType is guessed from the variable name. To override the guessed parameter type, specify a "parameter_type" meta column before calling interpolate. The guessed parameter types are returned in meta.

Parameters

- **target_times** (Union[ndarray, List[Union[datetime, int]]]) – Time grid onto which to interpolate
- **interpolation_type** (Union[*InterpolationType*, str]) – How to interpolate the data between timepoints
- **extrapolation_type** (Union[*ExtrapolationType*, str]) – If and how to extrapolate the data beyond the data in self.timeseries()

Returns A new *ScmDataFrameBase* containing the data interpolated onto the `target_times` grid

Return type *ScmDataFrameBase*

line_plot(*x='time'*, *y='value'*, ***kwargs*)
Plot a line chart.

See `pyam.IamDataFrame.line_plot()` for more information.

Return type `None`

property meta
Metadata

Return type `DataFrame`

pivot_table(*index*, *columns*, ***kwargs*)
Pivot the underlying data series.

See `pyam.core.IamDataFrame.pivot_table()` for details.

Return type `DataFrame`

process_over(*cols*, *operation*, ***kwargs*)
Process the data over the input columns.

Parameters

- **cols** (`Union[str, List[str]]`) – Columns to perform the operation on. The timeseries will be grouped by all other columns in *meta*.
- **operation** (*['median', 'mean', 'quantile']*) – The operation to perform. This uses the equivalent pandas function. Note that quantile means the value of the data at a given point in the cumulative distribution of values at each point in the timeseries, for each timeseries once the groupby is applied. As a result, using `q=0.5` is is the same as taking the median and not the same as taking the mean/average.
- ****kwargs** – Keyword arguments to pass to the pandas operation

Returns The quantiles of the timeseries, grouped by all columns in *meta* other than `cols`

Return type `pd.DataFrame`

Raises **ValueError** – If the operation is not one of [‘median’, ‘mean’, ‘quantile’]

region_plot(***kwargs*)
Plot regional data for a single model, scenario, variable, and year.

See `pyam.plotting.region_plot` for details.

Return type `None`

relative_to_ref_period_mean(*append_str=None*, ***kwargs*)
Return the timeseries relative to a given reference period mean.

The reference period mean is subtracted from all values in the input timeseries.

Parameters

- **append_str** (`Optional[str]`) – String to append to the name of all the variables in the resulting DataFrame to indicate that they are relevant to a given reference period. E.g. *‘rel. to 1961-1990’*. If None, this will be autofilled with the keys and ranges of `kwargs`.
- ****kwargs** – Arguments to pass to *filter()* to determine the data to be included in the reference time period. See the docs of *filter()* for valid options.

Returns DataFrame containing the timeseries, adjusted to the reference period mean

Return type `pd.DataFrame`

rename(*mapping*, *inplace=False*)

Rename and aggregate column entries using `groupby.sum()` on values. When renaming models or scenarios, the uniqueness of the index must be maintained, and the function will raise an error otherwise.

Parameters

- **mapping** (`Dict[str, Dict[str, str]]`) – For each column where entries should be renamed, provide current name and target name

```
{<column name>: {<current_name_1>: <target_name_1>,
                 <current_name_2>: <target_name_2>}}
```

- **inplace** (`bool`) – If `True`, do operation inplace and return `None`

Returns If `inplace` is `True`, return a new *ScmDataFrameBase* instance

Return type *ScmDataFrameBase*

Raises **ValueError** – Column is not in meta or renaming will cause non-unique metadata

resample(*rule='AS'*, ***kwargs*)

Resample the time index of the timeseries data onto a custom grid.

This helper function allows for values to be easily interpolated onto annual or monthly timesteps using the rules='AS' or 'MS' respectively. Internally, the interpolate function performs the regridding.

Parameters

- **rule** (`str`) – See the pandas user guide for a list of options. Note that Business-related offsets such as "BusinessDay" are not supported.
- ****kwargs** – Other arguments to pass through to *interpolate()*

Returns New *ScmDataFrameBase* instance on a new time index

Return type *ScmDataFrameBase*

Examples

Resample a dataframe to annual values

```
>>> scm_df = ScmDataFrame(
...     pd.Series([1, 2, 10], index=(2000, 2001, 2009)),
...     columns={
...         "model": ["a_iam"],
...         "scenario": ["a_scenario"],
...         "region": ["World"],
...         "variable": ["Primary Energy"],
...         "unit": ["EJ/y"],
...     }
... )
>>> scm_df.timeseries().T
model             a_iam
scenario     a_scenario
region            World
variable Primary Energy
unit               EJ/y
```

(continues on next page)

(continued from previous page)

```
year
2000     1
2010    10
```

An annual timeseries can be the created by interpolating to the start of years using the rule 'AS'.

```
>>> res = scm_df.resample('AS')
>>> res.timeseries().T
model                          a_iam
scenario                 a_scenario
region                        World
variable            Primary Energy
unit                            EJ/y
time
2000-01-01 00:00:00        1.000000
2001-01-01 00:00:00        2.001825
2002-01-01 00:00:00        3.000912
2003-01-01 00:00:00        4.000000
2004-01-01 00:00:00        4.999088
2005-01-01 00:00:00        6.000912
2006-01-01 00:00:00        7.000000
2007-01-01 00:00:00        7.999088
2008-01-01 00:00:00        8.998175
2009-01-01 00:00:00       10.00000
```

```
>>> m_df = scm_df.resample('MS')
>>> m_df.timeseries().T
model                          a_iam
scenario                 a_scenario
region                        World
variable            Primary Energy
unit                            EJ/y
time
2000-01-01 00:00:00        1.000000
2000-02-01 00:00:00        1.084854
2000-03-01 00:00:00        1.164234
2000-04-01 00:00:00        1.249088
2000-05-01 00:00:00        1.331204
2000-06-01 00:00:00        1.416058
2000-07-01 00:00:00        1.498175
2000-08-01 00:00:00        1.583029
2000-09-01 00:00:00        1.667883
                              ...
2008-05-01 00:00:00        9.329380
2008-06-01 00:00:00        9.414234
2008-07-01 00:00:00        9.496350
2008-08-01 00:00:00        9.581204
2008-09-01 00:00:00        9.666058
2008-10-01 00:00:00        9.748175
2008-11-01 00:00:00        9.833029
2008-12-01 00:00:00        9.915146
2009-01-01 00:00:00       10.000000
[109 rows x 1 columns]
```

Note that the values do not fall exactly on integer values as not all years are exactly the same length.

References

See the pandas documentation for *resample <http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.resample.html>* for more information about possible arguments.

scatter(*x*, *y*, ***kwargs*)
Plot a scatter chart using metadata columns.

See `pyam.plotting.scatter()` for details.

Return type `None`

set_meta(*meta*, *name=None*, *index=None*)
Set metadata information.

TODO: re-write this to make it more sane and add type annotations

Parameters

- **meta** (`Union[Series, list, int, float, str]`) – Column to be added to metadata
- **name** (`Optional[str]`) – Meta column name (defaults to `meta.name`)
- **index** (`Union[DataFrame, Series, Index, MultiIndex, None]`) – The index to which the metadata is to be applied

Raises **ValueError** – No name can be determined from inputs or index cannot be coerced to `pd.MultiIndex`

Return type `None`

tail(**args*, ***kwargs*)
Return tail of `self.timeseries()`.

Parameters

- ***args** – Passed to `self.timeseries().tail()`
- ****kwargs** – Passed to `self.timeseries().tail()`

Returns Tail of `self.timeseries()`

Return type `pd.DataFrame`

property time_points
Time points of the data

Return type `ndarray`

timeseries(*meta=None*)
Return the data in wide format (same as the timeseries method of `pyam.IamDataFrame`).

Parameters **meta** (`Optional[List[str]]`) – The list of meta columns that will be included in the output's MultiIndex. If None (default), then all metadata will be used.

Returns DataFrame with datetimes as columns and timeseries as rows. Metadata is in the index.

Return type `pd.DataFrame`

Raises **ValueError** – If the metadata are not unique between timeseries

to_csv(*path*, ***kwargs*)
Write timeseries data to a csv file

Parameters **path** (`str`) – Path to write the file into

Return type `None`

to_iamdataframe()
Convert to a LongDatetimeIamDataFrame instance.

LongDatetimeIamDataFrame is a subclass of pyam.IamDataFrame. We use LongDatetimeIamDataFrame to ensure all times can be handled, see docstring of LongDatetimeIamDataFrame for details.

Returns LongDatetimeIamDataFrame instance containing the same data.

Return type LongDatetimeIamDataFrame

Raises **ImportError** – If pyam is not installed

to_parameterset(*parameterset=None*)
Add parameters in this *ScmDataFrameBase* to a ParameterSet.

It can only be transformed if all timeseries have the same metadata. This is typically the case if all data comes from a single scenario/model input dataset. If that is not the case, further filtering is needed to reduce to a dataframe with identical metadata.

Parameters **parameterset** (Optional[ParameterSet]) – ParameterSet to add this *ScmDataFrameBase*'s parameters to. A new ParameterSet is created if this is None.

Returns ParameterSet containing the data in self (equals parameterset if not None)

Return type ParameterSet

Raises **ValueError** – Not all timeseries have the same metadata or climate_model is given and does not equal "unspecified"

property values
Timeseries values without metadata

Calls *timeseries()*

Return type ndarray

openscm.scmdataframe.base.**_format_data**(*df*)
Prepare data to initialize *ScmDataFrameBase* from pd.DataFrame or pd.Series.

See docstring of *ScmDataFrameBase.__init__()* for details.

Parameters **df** (Union[DataFrame, Series]) – Data to format.

Returns First dataframe is the data. Second dataframe is metadata.

Return type pd.DataFrame, pd.DataFrame

Raises **ValueError** – Not all required metadata columns are present or the time axis cannot be understood

openscm.scmdataframe.base.**_format_long_data**(*df*)

openscm.scmdataframe.base.**_format_wide_data**(*df*)

openscm.scmdataframe.base.**_from_ts**(*df*, *index=None*, ***columns*)
Prepare data to initialize *ScmDataFrameBase* from wide timeseries.

See docstring of *ScmDataFrameBase.__init__()* for details.

Returns First dataframe is the data. Second dataframe is metadata

Return type Tuple[pd.DataFrame, pd.DataFrame]

Raises **ValueError** – Not all required columns are present

openscm.scmdataframe.base.**_handle_potential_duplicates_in_append**(*data*, *duplicate_msg*)

openscm.scmdataframe.base.**_read_file**(*fnames*, **args*, ***kwargs*)
Prepare data to initialize *ScmDataFrameBase* from a file.

Parameters

- ***args** – Passed to *_read_pandas()*.
- ****kwargs** – Passed to *_read_pandas()*.

Returns First dataframe is the data. Second dataframe is metadata

Return type pd.DataFrame, pd.DataFrame

openscm.scmdataframe.base.**_read_pandas**(*fname*, **args*, ***kwargs*)
Read a file and return a pd.DataFrame.

Parameters

- **fname** (str) – Path from which to read data
- ***args** – Passed to pd.read_csv() if fname ends with '.csv', otherwise passed to pd.read_excel().
- ****kwargs** – Passed to pd.read_csv() if fname ends with '.csv', otherwise passed to pd.read_excel().

Returns Read data

Return type pd.DataFrame

Raises **OSError** – Path specified by fname does not exist

openscm.scmdataframe.base.**df_append**(*dfs*, *inplace=False*, *duplicate_msg='warn'*)
Append together many objects.

When appending many objects, it may be more efficient to call this routine once with a list of ScmDataFrames, than using ScmDataFrame.append() multiple times. If timeseries with duplicate metadata are found, the timeseries are appended and values falling on the same timestep are averaged (this behaviour can be adjusted with the duplicate_msg arguments).

Parameters

- **dfs** (List[Union[*ScmDataFrameBase*, None, DataFrame, Series, ndarray, str]]) – The dataframes to append. Values will be attempted to be cast to *ScmDataFrameBase*.
- **inplace** (bool) – If True, then the operation updates the first item in dfs and returns None.
- **duplicate_msg** (Union[str, bool]) – If "warn", raise a warning if duplicate data is detected. If "return", return the joint dataframe (including duplicate timeseries) so the user can inspect further. If False, take the average and do not raise a warning.

Returns If not inplace, the return value is the object containing the merged data. The resultant class will be determined by the type of the first object. If duplicate_msg == "return", a *pd.DataFrame* will be returned instead.

Return type *ScmDataFrameBase*

Raises

- **TypeError** – If inplace is True but the first element in dfs is not an instance of *ScmDataFrameBase*
- **ValueError** – duplicate_msg option is not recognised.

3.12.3 Filters

Helpers for filtering DataFrames.

Borrowed from `pyam.utils`.

`openscm.scmdataframe.filters.`**`datetime_match`**(*data*, *dts*)
Match datetimes in time columns for data filtering.

Parameters

- **`data`** (`List`[~T]) – Input data to perform filtering on
- **`dts`** (`Union`[`List`[`datetime`], `datetime`]) – Datetimes to use for filtering

Returns Array where `True` indicates a match

Return type `np.array` of `bool`

Raises **`TypeError`** – `dts` contains `int`

`openscm.scmdataframe.filters.`**`day_match`**(*data*, *days*)
Match days in time columns for data filtering.

Parameters

- **`data`** (`List`[~T]) – Input data to perform filtering on
- **`days`** (`Union`[`List`[`str`], `List`[`int`], `int`, `str`]) – Days to match

Returns Array where `True` indicates a match

Return type `np.array` of `bool`

`openscm.scmdataframe.filters.`**`find_depth`**(*meta_col*, *s*, *level*, *separator='|'*)
Find all values which match given depth from a filter keyword.

Parameters

- **`meta_col`** (`Series`) – Column in which to find values which match the given depth
- **`s`** (`str`) – Filter keyword, from which level should be applied
- **`level`** (`Union`[`int`, `str`]) – Depth of value to match as defined by the number of separator in the value name. If an int, the depth is matched exactly. If a str, then the depth can be matched as either "X-", for all levels up to level "X", or "X+", for all levels above level "X".
- **`separator`** (`str`) – The string used to separate levels in s. Defaults to a pipe ("|").

Returns Array where `True` indicates a match

Return type `np.array` of `bool`

Raises **`ValueError`** – If `level` cannot be understood

`openscm.scmdataframe.filters.`**`hour_match`**(*data*, *hours*)
Match hours in time columns for data filtering.

Parameters

- **`data`** (`List`[~T]) – Input data to perform filtering on
- **`hours`** (`Union`[`List`[`int`], `int`]) – Hours to match

Returns Array where `True` indicates a match

Return type `np.array` of `bool`

openscm.scmdataframe.filters.**is_in**(*vals*, *items*)
Find elements of vals which are in items.

Parameters

- **vals** (List[~T]) – The list of values to check
- **items** (List[~T]) – The options used to determine whether each element of vals is in the desired subset or not

Returns Array of the same length as vals where the element is True if the corresponding element of vals is in items and False otherwise

Return type np.array of bool

openscm.scmdataframe.filters.**month_match**(*data*, *months*)
Match months in time columns for data filtering.

Parameters

- **data** (List[~T]) – Input data to perform filtering on
- **months** (Union[List[str], List[int], int, str]) – Months to match

Returns Array where True indicates a match

Return type np.array of bool

openscm.scmdataframe.filters.**pattern_match**(*meta_col*, *values*, *level=None*, *regexp=False*, *has_nan=True*, *separator='|'*)
Filter data by matching metadata columns to given patterns.

Parameters

- **meta_col** (Series) – Column to perform filtering on
- **values** (Union[Iterable[str], str]) – Values to match
- **level** (Union[str, int, None]) – Passed to *find_depth()*. For usage, see docstring of *find_depth()*.
- **regexp** (bool) – If True, match using regexp rather than pseudo regexp syntax of pyam.
- **has_nan** (bool) – If True, convert all nan values in meta_col to empty string before applying filters. This means that "" and "*" will match rows with np.nan. If False, the conversion is not applied and so a search in a string column which contains np.nan will result in a TypeError.
- **separator** (str) – String used to separate the hierarchy levels in values. Defaults to '|'

Returns Array where True indicates a match

Return type np.array of bool

Raises **TypeError** – Filtering is performed on a string metadata column which contains np.nan and has_nan is False

openscm.scmdataframe.filters.**time_match**(*data*, *times*, *conv_codes*, *strptime_attr*, *name*)
Match times by applying conversion codes to filtering list.

Parameters

- **data** (List[~T]) – Input data to perform filtering on
- **times** (Union[List[str], List[int], int, str]) – Times to match
- **conv_codes** (List[str]) – If times contains strings, conversion codes to try passing to time.strptime() to convert times to datetime.datetime

- **strptime_attr** (str) – If times contains strings, the datetime.datetime attribute to finalize the conversion of strings to integers
- **name** (str) – Name of the part of a datetime to extract, used to produce useful error messages.

Returns Array where True indicates a match

Return type np.array of bool

Raises **ValueError** – If input times cannot be converted understood or if input strings do not lead to increasing integers (i.e. "Nov-Feb" will not work, one must use ["Nov-Dec", "Jan-Feb"] instead)

openscm.scmdataframe.filters.**years_match**(*data*, *years*)

Match years in time columns for data filtering.

Parameters

- **data** (List[~T]) – Input data to perform filtering on
- **years** (Union[List[int], int]) – Years to match

Returns Array where True indicates a match

Return type np.array of bool

Raises **TypeError** – If years is not int or list of int

3.12.4 Offsets

A simplified version of pandas.DateOffset`s which use datetime-like objects instead of :class:`pandas.Timestamp.

This differentiation allows for times which exceed the range of :class'pandas.Timestamp' (see here) which is particularly important for longer running models.

TODO: use np.timedelta64 instead?

openscm.scmdataframe.offsets.**apply_dt**(*func*, *self*)

Apply a wrapper which keeps the result as a datetime instead of converting to pd.Timestamp.

This decorator is a simplified version of pandas.tseries.offsets.apply_wraps(). It is required to avoid running into errors when our time data is outside panda's limited time range of 1677-09-22 00:12:43.145225 to 2262-04-11 23:47:16.854775807, see this discussion.

openscm.scmdataframe.offsets.**apply_rollback**(*obj*)

Roll provided date backward to previous offset, only if not on offset.

openscm.scmdataframe.offsets.**apply_rollforward**(*obj*)

Roll provided date forward to next offset, only if not on offset.

openscm.scmdataframe.offsets.**generate_range**(*start*, *end*, *offset*)

Generate a range of datetime objects between start and end, using offset to determine the steps.

The range will extend both ends of the span to the next valid timestep, see examples.

Parameters

- **start** (datetime) – Starting datetime from which to generate the range (noting roll backward mentioned above and illustrated in the examples).
- **end** (datetime) – Last datetime from which to generate the range (noting roll forward mentioned above and illustrated in the examples).

- **offset** (DateOffset) – Offset object for determining the timesteps. An offsetter obtained from :func‘to_offset‘ *must* be used.

Yields datetime.datetime – Next datetime in the range

Raises **ValueError** – Offset does not result in increasing :class‘datetime.datetime‘s

Examples

The range is extended at either end to the nearest timestep. In the example below, the first timestep is rolled back to 1st Jan 2001 whilst the last is extended to 1st Jan 2006.

```
>>> import datetime as dt
>>> from pprint import pprint
>>> from openscm.scmdataframe.offsets import to_offset, generate_range
>>> g = generate_range(
...     dt.datetime(2001, 4, 1),
...     dt.datetime(2005, 6, 3),
...     to_offset("AS"),
... )
```

```
>>> pprint([d for d in g])
[datetime.datetime(2001, 1, 1, 0, 0),
 datetime.datetime(2002, 1, 1, 0, 0),
 datetime.datetime(2003, 1, 1, 0, 0),
 datetime.datetime(2004, 1, 1, 0, 0),
 datetime.datetime(2005, 1, 1, 0, 0),
 datetime.datetime(2006, 1, 1, 0, 0)]
```

In this example the first timestep is rolled back to 31st Dec 2000 whilst the last is extended to 31st Dec 2005.

```
>>> g = generate_range(
...     dt.datetime(2001, 4, 1),
...     dt.datetime(2005, 6, 3),
...     to_offset("A"),
... )
>>> pprint([d for d in g])
[datetime.datetime(2000, 12, 31, 0, 0),
 datetime.datetime(2001, 12, 31, 0, 0),
 datetime.datetime(2002, 12, 31, 0, 0),
 datetime.datetime(2003, 12, 31, 0, 0),
 datetime.datetime(2004, 12, 31, 0, 0),
 datetime.datetime(2005, 12, 31, 0, 0)]
```

In this example the first timestep is already on the offset so stays there, the last timestep is to 1st Sep 2005.

```
>>> g = generate_range(
...     dt.datetime(2001, 4, 1),
...     dt.datetime(2005, 6, 3),
...     to_offset("QS"),
... )
>>> pprint([d for d in g])
[datetime.datetime(2001, 4, 1, 0, 0),
 datetime.datetime(2001, 7, 1, 0, 0),
 datetime.datetime(2001, 10, 1, 0, 0),
 datetime.datetime(2002, 1, 1, 0, 0),
 datetime.datetime(2002, 4, 1, 0, 0),
```

(continues on next page)

(continued from previous page)

```
datetime.datetime(2002, 7, 1, 0, 0),
datetime.datetime(2002, 10, 1, 0, 0),
datetime.datetime(2003, 1, 1, 0, 0),
datetime.datetime(2003, 4, 1, 0, 0),
datetime.datetime(2003, 7, 1, 0, 0),
datetime.datetime(2003, 10, 1, 0, 0),
datetime.datetime(2004, 1, 1, 0, 0),
datetime.datetime(2004, 4, 1, 0, 0),
datetime.datetime(2004, 7, 1, 0, 0),
datetime.datetime(2004, 10, 1, 0, 0),
datetime.datetime(2005, 1, 1, 0, 0),
datetime.datetime(2005, 4, 1, 0, 0),
datetime.datetime(2005, 7, 1, 0, 0)]
```

Return type Iterable[datetime]

openscm.scmdataframe.offsets.**to_offset**(*rule*)

Return a wrapped DateOffset instance for a given rule.

The DateOffset class is manipulated to return datetimes instead of pd.Timestamp, allowing it to handle times outside panda's limited time range of 1677-09-22 00:12:43.145225 to 2262-04-11 23:47:16.854775807, see this discussion.

Parameters **rule** (str) – The rule to use to generate the offset. For options see pandas offset aliases.

Returns Wrapped DateOffset class for the given rule

Return type DateOffset

Raises **ValueError** – If unsupported offset rule is requested, e.g. all business related offsets

3.12.5 Pyam Compatibilty

Imports and classes required to ensure compatibility with Pyam is intelligently handled.

3.13 Adapter base class

class openscm.adapters.**Adapter**(*input_parameters*, *output_parameters*)

Bases: object

All model adapters in OpenSCM are implemented as subclasses of the openscm.adapter.Adapter base class.

Writing a model adapter provides a how-to on implementing an adapter.

A model adapter is responsible for requesting the expected input parameters (in the expected time format and units) for the particular SCM from a openscm.core.ParameterSet. It also runs its wrapped SCM and writes the output data back to a openscm.core.ParameterSet.

__init__(*input_parameters*, *output_parameters*)

Initialize.

Parameters

- **input_parameters** (ParameterSet) – Input parameter set to use

- **output_parameters** (ParameterSet) – Output parameter set to use

_abc_impl = <_abc_data object>

_current_time = None
Current time when using *step()*

abstract _initialize_model()
To be implemented by specific adapters.

Initialize the model. Called only once but as late as possible before a call to *_run()* or *_step()*.

Return type None

abstract _initialize_model_input()
To be implemented by specific adapters.

Initialize the model input. Called before the adapter is used in any way and at most once before a call to *_run()* or *_step()*.

Return type None

abstract _initialize_run_parameters()
To be implemented by specific adapters.

Initialize parameters for the run. Called before the adapter is used in any way and at most once before a call to *_run()* or *_step()*.

Return type None

_initialized = None
True if model has been initialized via *_initialize_model()*

_output = None
Output parameter set

_parameters = None
Input parameter set

abstract _reset()
To be implemented by specific adapters.

Reset the model to prepare for a new run. Called once after each call of *_run()* and to reset the model after several calls to *_step()*.

Return type None

abstract _run()
To be implemented by specific adapters.

Run the model over the full time range.

Return type None

abstract _shutdown()
To be implemented by specific adapters.

Shut the model down.

Return type None

abstract _step()
To be implemented by specific adapters.

Do a single time step.

Return type None

initialize_model_input()
Initialize the model input.

Called before the adapter is used in any way and at most once before a call to *run()* or *step()*.

Return type None

initialize_run_parameters()
Initialize parameters for the run.

Called before the adapter is used in any way and at most once before a call to *run()* or *step()*.

Return type None

reset()
Reset the model to prepare for a new run.

Called once after each call of *run()* and to reset the model after several calls to *step()*.

Return type None

run()
Run the model over the full time range.

Return type None

step()
Do a single time step.

Returns Current time

Return type np.datetime64

3.14 Core

3.14.1 Parameters

Parameter handling.

openscm.core.parameters.**HIERARCHY_SEPARATOR = '|'**
String used to define different levels in our data hierarchies

By default we follow pyam and use "|". In such a case, emissions of CO2 for energy from coal would be "Emissions|CO2|Energy|Coal".

class openscm.core.parameters.**ParameterInfo**(*parameter*)
Bases: object

Information for a *parameter*.

__init__(*parameter*)
Initialize.

Parameters **parameter** (*_Parameter*) – Parameter

_parameter = None
Parameter

property empty
Check if parameter is empty, i.e. has not yet been written to.

Return type bool

ensure()
Ensure that parameter is not empty.

Raises ***ParameterEmptyError*** – If parameter is empty

Return type None

property name
Hierarchical name of the parameter

Return type Tuple[str, ...]

property parameter_type
Parameter type

Return type Optional[*ParameterType*]

property region
Hierarchichal name of the region this parameter belongs to

Return type Tuple[str, ...]

property unit
Parameter unit

Return type Optional[str]

property version
Version number of parameter (used internally)

Return type int

class openscm.core.parameters.**ParameterType**
Bases: enum.Enum

Parameter type.

AVERAGE_TIMESERIES = 2

GENERIC = 4

POINT_TIMESERIES = 3

SCALAR = 1

from_timeseries_type(*timeseries_type*) **= <bound method ParameterType.**
from_timeseries_type of <enum 'ParameterType'>>

timeseries_type_to_string(*timeseries_type*) **= <bound method ParameterType.**
timeseries_type_to_string of <enum
'ParameterType'>>

class openscm.core.parameters.**_Parameter**(*name*, *region*)
Bases: object

Represents a *parameter* in the *parameter hierarchy*.

__init__(*name*, *region*)
Initialize.

Parameters **name** (str) – Name

attempt_read(*parameter_type*, *unit=None*, *time_points=None*)
Tell parameter that it will be read from. If the parameter has child parameters it will be read in in an aggregated way, i.e., aggregating over child parameters.

Parameters

- **parameter_type** (*ParameterType*) – Parameter type to be read
- **unit** (Optional[str]) – Unit to be read; only for scalar and timeseries parameters
- **time_points** (Optional[ndarray]) – Timeseries time points; only for timeseries parameters

Raises

- ***ParameterTypeError*** – If parameter has already been read from or written to in a different type
- ***ParameterAggregationError*** – If parameter has child parameters which cannot be aggregated (for boolean and string parameters)

Return type None

attempt_write(*parameter_type*, *unit=None*, *time_points=None*)
Tell parameter that its data will be written to.

Parameters

- **parameter_type** (*ParameterType*) – Parameter type to be written
- **unit** (Optional[str]) – Unit to be written; only for scalar and timeseries parameters
- **time_points** (Optional[ndarray]) – Timeseries time points; only for timeseries parameters

Raises ***ParameterReadonlyError*** – If parameter is read-only because it has child parameters

Return type None

children = None
Child parameters

data = None
Data

property full_name
Full *hierarchical name*

Return type Tuple[str, ...]

get_or_create_child_parameter(*name*)
Get a (direct) child parameter of this parameter. Create and add it if not found.

Parameters **name** (str) – Name

Returns Parameter found or newly created

Return type *_Parameter*

Raises

- ***ParameterReadError*** – If the child parameter would need to be added, but this parameter has already been read from. In this case a child parameter cannot be added.
- ***ParameterWrittenError*** – If the child parameter would need to be added, but this parameter has already been written to. In this case a child parameter cannot be added.

get_subparameter(*name*)
Get a sub parameter of this parameter or None if not found.

Parameters **name** (Union[str, Sequence[str]]) – *Hierarchical name* of the subparameter below this parameter or () for this parameter

Returns Parameter of `None` if not found

Return type Optional[*_Parameter*]

has_been_read_from = None
If True, parameter has already been read from

has_been_written_to = None
If True, parameter data has already been changed

name = None
Name

parameter_type = None
Parameter type

parent = None
Parent parameter

region = None
Region this parameter belongs to

time_points = None
Timeseries time points; only for timeseries parameters

unit = None
Unit

version = None
Internal version (incremented by each write operation)

3.14.2 Parameter views

Parameter views provide ways to read and write parameter data with a defined unit and time information.

class openscm.core.views.**GenericView**(*parameter*)
Bases: *openscm.core.parameters.ParameterInfo*

View of a generic parameter.

__init__(*parameter*)
Initialize.

Parameters **parameter** (*_Parameter*) – Parameter to handle

_writable = None
Is this view writable? Is set to `True` once this view is written to.

property empty
Check if parameter is empty, i.e. has not yet been written to.

Return type `bool`

ensure()
Ensure that parameter is not empty.

Raises ***ParameterEmptyError*** – If parameter is empty

Return type `None`

property name
Hierarchical name of the parameter

Return type `Tuple[str, ...]`

property parameter_type
Parameter type

Return type Optional[*ParameterType*]

property region
Hierarchichal name of the region this parameter belongs to

Return type Tuple[str, ...]

property unit
Parameter unit

Return type Optional[str]

property value
Value of generic parameter

Raises

- ***ParameterEmptyError*** – Parameter is empty, i.e. has not yet been written to
- ***ParameterReadonlyError*** – Parameter is read-only (e.g. because its parent has been written to)

Return type Any

property version
Version number of parameter (used internally)

Return type int

class openscm.core.views.**ScalarView**(*parameter*, *unit*)
Bases: *openscm.core.parameters.ParameterInfo*

View of a scalar parameter.

__init__(*parameter*, *unit*)
Initialize.

Parameters

- **parameter** (*_Parameter*) – Parameter to handle
- **unit** (str) – Unit for the values in the view

_child_data_views = None
List of views to the child parameters for aggregated reads

_unit_converter = None
Unit converter

_writable = None
Is this view writable? Is set to True once this view is written to.

property empty
Check if parameter is empty, i.e. has not yet been written to.

Return type bool

ensure()
Ensure that parameter is not empty.

Raises ***ParameterEmptyError*** – If parameter is empty

Return type None

property name
Hierarchical name of the parameter

Return type Tuple[str, ...]

property parameter_type
Parameter type

Return type Optional[*ParameterType*]

property region
Hierarchichal name of the region this parameter belongs to

Return type Tuple[str, ...]

property unit
Parameter unit

Return type Optional[str]

property value
Value of scalar parameter

If the parameter has child parameters, the returned value will be the sum of the values of all of the child parameters.

Raises

- ***ParameterEmptyError*** – Parameter is empty, i.e. has not yet been written to
- ***ParameterReadonlyError*** – Parameter is read-only (e.g. because its parent has been written to)

Return type float

property version
Version number of parameter (used internally)

Return type int

class openscm.core.views.**TimeseriesView**(*parameter*, *unit*, *time_points*, *timeseries_type*, *interpolation_type*, *extrapolation_type*)

Bases: *openscm.core.parameters.ParameterInfo*

View of a timeseries.

__init__(*parameter*, *unit*, *time_points*, *timeseries_type*, *interpolation_type*, *extrapolation_type*)
Initialize.

Parameters

- **parameter** (*_Parameter*) – Parameter
- **unit** (str) – Unit for the values in the view
- **time_points** (ndarray) – Timeseries time points
- **timeseries_type** (*ParameterType*) – Time series type
- **interpolation_type** (*InterpolationType*) – Interpolation type
- **extrapolation_type** (*ExtrapolationType*) – Extrapolation type

_check_write()

_child_data_views = None
List of views to the child parameters for aggregated reads

_data = None
Chache for underlying data

_get_values()

Return type ndarray

_locked = None
Is this view locked (i.e., does it not update the underlying parameter on every write)?

_read()

_timeseries = None
Time series handler

_timeseries_converter = None
Timeseries converter

_unit_converter = None
Unit converter

_version = None
Version of cache

_writable = None
Is this view writable? Is set to `True` once this view is written to.

_write()

property empty
Check if parameter is empty, i.e. has not yet been written to.

Return type bool

ensure()
Ensure that parameter is not empty.

Raises ***ParameterEmptyError*** – If parameter is empty

Return type None

property length
Length of timeseries

Return type int

lock()
Lock this view (i.e., do not update the underlying parameter on every write).

Return type None

property name
Hierarchical name of the parameter

Return type Tuple[str, ...]

property parameter_type
Parameter type

Return type Optional[*ParameterType*]

property region
Hierarchichal name of the region this parameter belongs to

Return type Tuple[str, ...]

property unit
Parameter unit

Return type Optional[str]

unlock()
Unlock this view (i.e., update the underlying parameter on every write).

Updates the underlying parameter.

Return type None

property values
Values of the full timeseries

If the parameter has child parameters, the returned value will be the sum of the values of all of the child parameters.

Raises

- ***ParameterEmptyError*** – Parameter is empty, i.e. has not yet been written to
- ***ParameterReadonlyError*** – Parameter is read-only (e.g. because its parent has been written to)
- ***TimeseriesPointsValuesMismatchError*** – Lengths of set value and the time points number mismatch

Return type *_Timeseries*

property version
Version number of parameter (used internally)

Return type int

class openscm.core.views.**_Timeseries**(*input_array*, *parameter_view*)
Bases: pandas.core.arrays.base.ExtensionOpsMixin, numpy.lib.mixins.NDArrayOperatorsMixin

Internal class which wraps numpy to make sure data is buffered and up-to-date

_HANDLED_TYPES = (<class 'numpy.ndarray'>, <class 'numbers.Number'>)

__init__(*input_array*, *parameter_view*)
Initialize.

Parameters

- **input_array** (ndarray) – Array to handle
- **parameter_view** (*TimeseriesView*) – *TimeseriesView* this timeseries belongs to

classmethod _add_arithmetic_ops()

classmethod _add_comparison_ops()

classmethod _create_arithmetic_method(*op*)

classmethod _create_comparison_method(*op*)

property dtype
np.dtype of the timeseries

Return type dtype

property nbytes
Bytes block of the unterlying timeseries

property ndim
Dimension of the timeseries (==1)

property shape
Shape of the 1-dimensional timeseries array

Return type Tuple[int, ...]

3.14.3 Regions

Handling of region information.

class openscm.core.regions.**_Region**(*name*)
Bases: object

Represents a region in the region hierarchy.

__init__(*name*)
Initialize

Parameters **name** (str) – Name

_children = None
Subregions

_has_been_aggregated = None
If True, a parameter of this region has already been read in an aggregated way, i.e., aggregating over subregions

_name = None
Name

_parameters = None
Parameters

_parent = None
Parent region (or None if root region)

attempt_aggregate()
Tell region that one of its parameters will be read from in an aggregated way, i.e., aggregating over subregions.

Return type None

property full_name
Full hierarchical name

Return type Tuple[str, ...]

get_or_create_parameter(*name*)
Get a root parameter for this region. Create and add it if not found.

Parameters **name** (str) – Name

Returns Root parameter found or newly created

Return type *parameters._Parameter*

get_or_create_subregion(*name*)
Get a (direct) subregion of this region. Create and add it if not found.

Parameters **name** (str) – Name

Returns Region found or newly created

Return type *_Region*

Raises ***RegionAggregatedError*** – If the subregion would need to be added and a parameter of this region has already been read in an aggregated way. In this case a subregion cannot be created.

get_parameter(*name*)
Get a (root or sub-) parameter for this region or None if not found.

Parameters **name** (Union[str, Sequence[str]]) – *Hierarchical name* of the parameter

Returns Parameter of None if not found

Return type Optional[*parameters._Parameter*]

Raises **ValueError** – Name not given

get_subregion(*name*)
Get a subregion of this region or None if not found.

Parameters **name** (Union[str, Sequence[str]]) – Hierarchical name of the region below this region or () for this region

Returns Subregion or None if not found

Return type Optional[*_Region*]

property name
Name

Return type str

property parent
Parent region (or None if root region)

Return type Optional[*_Region*]

3.14.4 Time

Different climate models often use different time frames for their input and output data. This includes different 'meanings' of time steps (e.g. beginning vs middle of year) and different lengths of the time steps (e.g. years vs months). Accordingly, OpenSCM supports the conversion of timeseries data between such timeseries, which is handled in this module. A thorough explaination of the procedure used is given in a dedicated Jupyter Notebook.

class openscm.core.time.**ExtrapolationType**
Bases: enum.Enum

Extrapolation type.

CONSTANT = 0

LINEAR = 1

NONE = -1

from_extrapolation_type(*extrapolation_type*) **= <bound method ExtrapolationType.from_extrapolation_type of <enum 'ExtrapolationType'>>**

class openscm.core.time.**InterpolationType**
Bases: enum.Enum

Interpolation type.

LINEAR = 1

from_interpolation_type(*interpolation_type*) **= <bound method InterpolationType.from_interpolation_type of <enum 'InterpolationType'>>**

class openscm.core.time.**TimePoints**(*values*)
Bases: object

Handles time points by wrapping np.ndarray of np.datetime64..

__init__(*values*)
Initialize.

Parameters **values** (Sequence[+T_co]) – Time points array to handle

_values = None
Actual time points array

days()
Get day of each time point.

Returns Day of each time point

Return type np.array of int

hours()
Get hour of each time point.

Returns Hour of each time point

Return type np.array of int

months()
Get month of each time point.

Returns Month of each time point

Return type np.array of int

to_index()
Get time points as pd.Index.

Returns pd.Index of np.dtype object with name "time" made from the time points represented as datetime.datetime.

Return type pd.Index

property values
Time points

Return type ndarray

weekdays()
Get weekday of each time point.

Returns Day of the week of each time point

Return type np.array of int

years()
Get year of each time point.

Returns Year of each time point

Return type np.array of int

class openscm.core.time.**TimeseriesConverter**(*source_time_points*, *target_time_points*, *timeseries_type*, *interpolation_type*, *extrapolation_type*)

Bases: object

Converts timeseries and their points between two timeseriess (each defined by a time of the first point and a period length).

__init__(*source_time_points*, *target_time_points*, *timeseries_type*, *interpolation_type*, *extrapolation_type*)

Initialize.

Parameters

- **source_time_points** (ndarray) – Source timeseries time points
- **target_time_points** (ndarray) – Target timeseries time points
- **timeseries_type** (*ParameterType*) – Time series type
- **interpolation_type** (*InterpolationType*) – Interpolation type
- **extrapolation_type** (*ExtrapolationType*) – Extrapolation type

Raises ***InsufficientDataError*** – Timeseries too short to extrapolate

_calc_continuous_representation(*time_points*, *values*)

Calculate a "continuous" representation of a timeseries (see openscm.timeseries_converter._calc_integral_preserving_linear_interpolation()) with the time points time_points and values values.

Parameters

- **time_points** (ndarray) – Time points of the timeseries
- **values** (ndarray) – Values of the timeseries

Returns Function that represents the interpolated timeseries. It takes a single argument, time ("x-value"), and returns a single float, the value of the interpolated timeseries at that point in time ("y-value").

Return type Callable[[float], float]

_convert(*values*, *source_time_points*, *target_time_points*)

Wrap *_convert_unsafe()* to provide proper error handling.

_convert_unsafe() converts time period average timeseries data values for timeseries time points source_time_points to the time points target_time_points.

Parameters

- **values** (ndarray) – Array of data to convert
- **source_time_points** (ndarray) – Source timeseries time points
- **target_time_points** (ndarray) – Target timeseries time points

Raises

- ***InsufficientDataError*** – Length of the time series is too short to convert
- ***InsufficientDataError*** – Target time points are outside the source time points and *_extrapolation_type* is *ExtrapolationType.NONE*

Returns Converted time period average data for timeseries `values`

Return type np.ndarray

_convert_unsafe(*values*, *source_time_points*, *target_time_points*)
Convert time period average timeseries data `values` for timeseries time points `source_time_points` to the time points `target_time_points`.

Parameters

- **values** (ndarray) – Array of data to convert
- **source_time_points** (ndarray) – Source timeseries time points
- **target_time_points** (ndarray) – Target timeseries time points

Raises **NotImplementedError** – The timeseries type is not recognised

Returns Converted time period average data for timeseries `values`

Return type np.ndarray

_extrapolation_type = None
Extrapolation type

_get_scipy_extrapolation_args(*values*)

Return type Dict[str, Any]

_get_scipy_interpolation_arg()

Return type str

_interpolation_type = None
Interpolation type

_source = None
Source timeseries time points

_target = None
Target timeseries time points

_timeseries_type = None
Time series type

convert_from(*values*)
Convert value **from** source timeseries time points to target timeseries time points.

Parameters **values** (ndarray) – Value

Returns Converted array

Return type np.ndarray

convert_to(*values*)
Convert value from target timeseries time points **to** source timeseries time points.

Parameters **values** (ndarray) – Value

Returns Converted array

Return type np.ndarray

property source_length
Length of source timeseries

Return type int

property target_length
Length of target timeseries

Return type int

openscm.core.time.**_calc_integral_preserving_linear_interpolation**(*values*)
Calculate the "linearization" values of the array values which is assumed to represent averages over time periods. Values at the edges of the periods are taken as the average of adjacent periods, values at the period middles are taken such that the integral over a period is the same as for the input data.

Parameters **values** (ndarray) – Timeseries values of period averages

Returns Values of linearization (of length 2 * len(values) + 1)

Return type np.ndarray

openscm.core.time.**_calc_interval_averages**(*continuous_representation*, *target_intervals*)
Calculate the interval averages of a continuous function.

Here interval average is calculated as the integral over the period divided by the period length.

Parameters

- **continuous_representation** (Callable[[float], float]) – Continuous function from which to calculate the interval averages. Should be calculated using openscm.timeseries_converter.TimeseriesConverter._calc_continuous_representation().
- **target_intervals** (ndarray) – Intervals to calculate the average of.

Returns Array of the interval/period averages

Return type np.ndarray

openscm.core.time.**_float_year_to_datetime**(*inp*)

Return type datetime64

openscm.core.time.**_format_datetime**(*dts*)
Convert an array to an array of np.datetime64.

Parameters **dts** (ndarray) – Input to attempt to convert

Returns Converted array

Return type np.ndarray of np.datetime64

Raises **ValueError** – If one of the values in dts cannot be converted to np.datetime64

openscm.core.time.**_parse_datetime**(*inp*)

Return type ndarray

openscm.core.time.**create_time_points**(*start_time*, *period_length*, *points_num*, *timeseries_type*)
Create time points for an equi-distant time series.

Parameters

- **start_time** (datetime64) – First time point of the timeseries
- **period_length** (timedelta64) – Period length
- **points_num** (int) – Length of timeseries
- **timeseries_type** (Union[*ParameterType*, str]) – Timeseries type

Returns Array of the timeseries time points

Return type `np.ndarray` of `np.datetime64`

3.15 Units

Unit handling.

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.

In general, you should not use Pint with OpenSCM explicitly. As illustration of how units are used internally, we provide the following example:

```
>>> 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 C / week")
<Quantity(6.516224050620789, 'C * megametric_ton / week')>
```

A note 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', we provide conversions
- 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" but 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).

Finally, contexts are particularly useful for emissions as they facilitate much easier 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 UnitConverter
>>> uc = UnitConverter("CH4", "C")
pint.errors.DimensionalityError: Cannot convert from 'CH4' ([methane]) to 'C'␣
↪([carbon])

# with a context, the conversion becomes legal again
>>> uc = UnitConverter("CH4", "C", context="CH4_conversions")
>>> uc.convert_from(1)
0.75

# as an unavoidable side effect, this also becomes possible
>>> uc = UnitConverter("CH4", "CO2", context="CH4_conversions")
>>> uc.convert_from(1)
2.75
```

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
>>> uc = UnitConverter("NOx", "N")
pint.errors.DimensionalityError: Cannot convert from 'NOx' ([NOx]) to 'N' ([nitrogen])

# with a context, the conversion becomes legal again
>>> uc = UnitConverter("NOx", "N", context="NOx_conversions")
>>> uc.convert_from(1)
0.30434782608695654

# as an unavoidable side effect, this also becomes possible
>>> uc = UnitConverter("NOx", "N2O", context="NOx_conversions")
>>> uc.convert_from(1)
0.9565217391304348
```

class openscm.core.units.**ScmUnitRegistry**(*filename=''*, *force_ndarray=False*, *default_as_delta=True*, *autoconvert_offset_to_baseunit=False*, *on_redefinition='warn'*, *system=None*, *auto_reduce_dimensions=False*)

Bases: `pint.registry.UnitRegistry`

Unit registry class for OpenSCM. Provides some convenience methods to add standard unit and contexts.

__init__(*filename=''*, *force_ndarray=False*, *default_as_delta=True*, *autoconvert_offset_to_baseunit=False*, *on_redefinition='warn'*, *system=None*, *auto_reduce_dimensions=False*)

Initialize self. See help(type(self)) for accurate signature.

_add_gases(*gases*)

Return type `None`

_add_mass_emissions_joint_version(*symbol*)

Add a unit which is the combination of mass and emissions.

This allows users to units like e.g. `"tC"` rather than requiring a space between the mass and the emissions i.e. `"t C"`

Parameters **symbol** (str) – The unit to add a joint version for

Return type None

_after_init()
This should be called after all __init__

_build_cache()
Build a cache of dimensionality and base units.

_contexts_loaded = False

_convert(*value*, *src*, *dst*, *inplace=False*)
Convert value from some source to destination units.

In addition to what is done by the BaseRegistry, converts between units with different dimensions by following transformation rules defined in the context.

Parameters

- **value** – value
- **src** (*UnitsContainer*) – source units.
- **dst** (*UnitsContainer*) – destination units.

Returns converted value

_dedup_candidates(*candidates*)
Given a list of unit triplets (prefix, name, suffix), remove those with different names but equal value.

e.g. (‘kilo’, ‘gram’, ‘’) and (‘’, ‘kilogram’, ‘’)

_define(*definition*)
Add unit to the registry.

This method defines only multiplicative units, converting any other type to *delta_* units.

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

Returns Definition instance, case sensitive unit dict, case insensitive unit dict.

Return type Definition, dict, dict

_define_adder(*definition*, *unit_dict*, *casei_unit_dict*)
Helper function to store a definition in the internal dictionaries. It stores the definition under its name, symbol and aliases.

_define_single_adder(*key*, *value*, *unit_dict*, *casei_unit_dict*)
Helper function to store a definition in the internal dictionaries.

It warns or raise error on redefinition.

_dir = ['Quantity', 'Unit', 'Measurement', 'define', 'load_definitions', 'get_name', '

_eval_token(*token*, *case_sensitive=True*, ***values*)

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

_get_compatible_units(*input_units*, *group_or_system*)

_get_dimensionality(*input_units*)
Convert a UnitsContainer to base dimensions.

Parameters **input_units** –

Returns dimensionality

_get_dimensionality_ratio(*unit1*, *unit2*)
Get the exponential ratio between two units, i.e. solve unit2 = unit1**x for x. :param unit1: first unit :type unit1: UnitsContainer compatible (str, Unit, UnitsContainer, dict) :param unit2: second unit :type unit2: UnitsContainer compatible (str, Unit, UnitsContainer, dict) :returns: exponential proportionality or None if the units cannot be converted

_get_dimensionality_recurse(*ref*, *exp*, *accumulator*)

_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 dict*) – units
- **check_nonmult** – if True, None will be returned as the multiplicative factor if a non-multiplicative units is found in the final Units.

Returns multiplicative factor, base units

_get_root_units_recurse(*ref*, *exp*, *accumulators*)

_get_symbol(*name*)

_is_multiplicative(*u*)

_load_contexts()
Load contexts.

Return type None

_load_metric_conversions()
Load metric conversion contexts from file.

This is done only when contexts are needed to avoid reading files on import.

Return type None

_parse_context(*ifile*)

_parse_defaults(*ifile*)
Loader for a @default section.

_parse_group(*ifile*)

_parse_system(*ifile*)

_parse_units(*input_string*, *as_delta=None*)

_parsers = None

_register_parser(*prefix*, *parserfunc*)
Register a loader for a given @ directive..

Parameters

- **prefix** – string identifying the section (e.g. @context)
- **parserfunc** (*SourceIterator -> None*) – A function that is able to parse a Definition section.

_register_parsers()

_validate_and_extract(*units*)

add_context(*context*)
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. Use *enable_contexts*.

add_standards()
Add standard units.

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

check(**args*)
Decorator to for quantity type checking for function inputs.

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

Use None to skip argument checking.

Parameters

- **ureg** – a UnitRegistry instance.
- **args** – iterable of input units.

Returns the wrapped function.

Raises `DimensionalityError` if the parameters don't match dimensions

context(**names*, ***kwargs*)
Used as a context manager, this function enables to activate a context which is removed after usage.

Parameters

- **names** – name of the context.
- **kwargs** – keyword arguments for the contexts.

Context are called by their name:

```
>>> with ureg.context('one'):
...     pass
```

If the 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):
...     with ureg.context('two'): # Here n takes the value of the upper␣
↪context
...         pass
```

convert (*value*, *src*, *dst*, *inplace=False*)
Convert value from some source to destination units.

Parameters

- **value** – value
- **src** (*Quantity or str*) – source units.
- **dst** (*Quantity or str*) – destination units.

Returns converted value

property default_format
Default formatting string for quantities.

property default_system

define (*definition*)
Add unit to the registry.

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

disable_contexts (*n=None*)
Disable the last n enabled 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.

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** – if True, None will be returned as the multiplicative factor if a non-multiplicative units is found in the final Units.

Returns multiplicative factor, base units

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

Parameters **input_units** –

Returns dimensionality

get_group (*name*, *create_if_needed=True*)
Return a Group.

Parameters

- **name** – Name of the group to be
- **create_if_needed** – Create a group if not Found. If False, raise an Exception.

Returns Group

get_name(*name_or_alias*, *case_sensitive=True*)
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** – if True, None will be returned as the multiplicative factor if a non-multiplicative units is found in the final Units.

Returns multiplicative factor, base units

get_symbol(*name_or_alias*)
Return the preferred alias for a unit

get_system(*name*, *create_if_needed=True*)
Return a Group.

Parameters

- **name** – Name of the group to be
- **create_if_needed** – Create a group if not Found. If False, raise an Exception.

Returns System

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.

parse_expression(*input_string*, *case_sensitive=True*, ***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.

parse_unit_name(*unit_name*, *case_sensitive=True*)
Parse a unit to identify prefix, unit name and suffix by walking the list of prefix and suffix.

Return type (str, str, str)

parse_units(*input_string*, *as_delta=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 **as_delta** – if the expression has multiple units, the parser will interpret non multiplicative units as their *delta_* counterparts.

Raises `pint.UndefinedUnitError` if a unit is not in the registry `ValueError` if the expression is invalid.

pi_theorem(*quantities*)
Builds dimensionless quantities using the Buckingham theorem :param quantities: mapping between variable name and units :type quantities: dict :return: a list of dimensionless quantities expressed as dicts

remove_context(*name_or_alias*)
Remove a context from the registry and return it.

Notice that this methods will not disable the context. Use *disable_contexts*.

setup_matplotlib(*enable=True*)
Set up handlers for matplotlib's unit support. :param enable: whether support should be enabled or disabled :type enable: bool

property sys

with_context(*name*, ***kw*)
Decorator to wrap a function call in a Pint context.

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

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

Parameters

- **names** – name of the context.
- **kwargs** – keyword arguments for the contexts.

Returns the wrapped function.

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*.

Use None to skip argument conversion. Set strict to False, to accept also numerical values.

Parameters

- **ureg** – a UnitRegistry instance.
- **ret** – output units.
- **args** – iterable of input units.
- **strict** – boolean to indicate that only quantities are accepted.

Returns the wrapped function.

Raises `ValueError` if strict and one of the arguments is not a Quantity.

class `openscm.core.units.`**UnitConverter**(*source*, *target*, *context=None*)
Bases: `object`

Converts numbers between two units.

__init__ (*source*, *target*, *context=None*)
Initialize.

Parameters

- **source** (str) – Unit to convert **from**
- **target** (str) – Unit to convert **to**
- **context** (Optional[str]) – Context to use for the conversion i.e. which metric to apply when performing CO2-equivalent calculations. If None, no metric will be applied and CO2-equivalent calculations will raise DimensionalityError.

Raises

- **pint.errors.DimensionalityError** – Units cannot be converted into each other.
- **pint.errors.UndefinedUnitError** – Unit undefined.

_offset = None
Offset for units (e.g. for temperature units)

_scaling = None
Scaling factor between units

_source = None
Source unit

_target = None
Target unit

property contexts
Available contexts for unit conversions

Return type Sequence[str]

convert_from(*v*)
Convert value **from** source unit to target unit.

Parameters **value** – Value in source unit

Returns Value in target unit

Return type Union[float, np.ndarray]

convert_to(*v*)
Convert value from target unit **to** source unit.

Parameters **value** – Value in target unit

Returns Value in source unit

Return type Union[float, np.ndarray]

property source
Source unit

Return type str

property target
Target unit

Return type str

property unit_registry
Underlying unit registry

Return type *ScmUnitRegistry*

openscm.core.units.**_unit_registry**(*input_string*, *case_sensitive=True*, ***values*) = **<openscm.core.units.ScmUnitRegistry object>**

OpenSCM standard unit registry

The unit registry contains all of the recognised units.

3.16 Changelog

3.16.1 master

- (#184) Fix unit view bug identified in #177
- (#146) Refactor the core interface
- (#168) Fix false positive detection of duplicates when appending timeseries
- (#166) Add usage guidelines
- (#165) Add `openscm.scenarios` module with commonly used scenarios
- (#163) Lock `pyam` version to pypi versions
- (#160) Update `setup.py` so project description is right
- (#158) Add `ScmDataFrame`, a high-level data and analysis class
- (#147) Remove pyam dependency
- (#142) Add boolean and string parameters
- (#140) Add SARGWP100, AR4GWP100 and AR5GWP100 conversion contexts
- (#139) Add initial definition of parameters
- (#138) Add support for linear point interpolation as well as linear and constant extrapolation
- (#134) Fix type annotations and add a static checker
- (#133) Cleanup and advance timeseries converter
- (#125) Renamed `timeframes` to `timeseries` and simplified interpolation calculation
- (#104) Define Adapter interface
- (#92) Updated installation to remove notebook dependencies from minimum requirements as discussed in #90
- (#87) Added aggregated read of parameter
- (#86) Made top level of region explicit, rather than allowing access via `()` and made requests robust to string inputs
- (#85) Split out submodule for ScmDataFrameBase `openscm.scmdataframebase` to avoid circular imports
- (#83) Rename OpenSCMDataFrame to ScmDataFrame
- (#78) Added OpenSCMDataFrame
- (#44) Add timeframes module
- (#40) Add parameter handling in core module
- (#35) Add units module

PYTHON MODULE INDEX

O

INDEX

Symbols

A

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Y