

---

# **SCMData Documentation**

***Release 0.13.0***

**Jared Lewis, Zebedee Nicholls**

**Sep 13, 2021**



## DOCUMENTATION:

<b>1</b>	<b>Brief summary</b>	<b>1</b>
1.1	License . . . . .	1
<b>2</b>	<b>Indices and tables</b>	<b>77</b>
	<b>Python Module Index</b>	<b>79</b>
	<b>Index</b>	<b>81</b>



## BRIEF SUMMARY

**scmdata** provides some useful data handling routines for dealing with data related to simple climate models (SCMs aka reduced complexity climate models, RCMs). In particular, it provides a high-performance way of handling and serialising (including to netCDF) timeseries data along with attached metadata. **scmdata** was inspired by [pyam](#) and was originally part of the [openscm](#) package.

### 1.1 License

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

#### 1.1.1 Installation

scmdata is tested to work with Python 3.6 and above

##### Installing with conda

The easiest way to install scmdata is using conda, either using the full [Anaconda](#) distribution which includes a collection of popular data science packages or the smaller [Miniconda](#) distribution. Using conda is the recommended method for installing scmdata for most users.

```
conda install -c conda-forge netcdf-scm
```

##### Installing with pip

scmdata can also be installed from [PyPi](#) using pip.

We recommend creating a virtual environment to manage this and any other libraries your project requires.

```
pip install scmdata
```

### 1.1.2 Data Model

Analysing the results from simple climate models involves a lot of timeseries handling, including:

- filtering
- plotting
- resampling
- serialization/deserialisation
- computation

As a result, **scmdata**'s approach to data handling focusses on efficient handling of timeseries.

#### The `ScmRun` class

The `scmdata.ScmRun` class represents a collection of timeseries data including metadata and provides methods for manipulating the data. Internally, `ScmRun` stores the timeseries data in a single `pandas.DataFrame` and the timeseries metadata `pandas.MultiIndex` of type `pandas.Categorical`, for efficient indexing.

This class is the primary way of handling timeseries data within the **scmdata** package. For example, the `ScmRun` can be filtered to only find the subset of data which have a "scenario" metadata label equal to "green" (see `ScmRun.filter` for full details). Other operations include grouping, setting and (basic) plotting.

The complete set of manipulation features can be found in the documentation pages of `ScmRun`.

`ScmRun` has three key properties and one key method, which allow the user to quickly access their data in more standard formats:

- `values` returns all of the timeseries as a single `numpy.ndarray` without any metadata or indication of the time axis.
- `meta` returns all of the timeseries' metadata as a single `pandas.DataFrame`. This allows users to quickly have an overview of the timeseries held by `scmdata.ScmRun` without having to also view the data itself.
- `metadata <scmdata.run.ScmRun.metadata` stores run-specific metadata, i.e. metadata which isn't tied to any timeseries specifically.
- `timeseries()` combines `values` and `meta` to form a `pandas.DataFrame` whose index is equal to `scmdata.ScmRun.meta` and whose values are equal to `scmdata.ScmRun.values`. The columns of the output of `timeseries()` are the time axis of the data.

#### Metadata handling

**scmdata** can store any kind of metadata about the timeseries, without restriction. This combination allows it to be a high performing, yet flexible library for timeseries data.

However, to do this it must make assumptions about the type of data it holds and these assumptions come with tradeoffs. In particular, **scmdata** cannot hold metadata at a level finer than a complete timeseries. For example, it couldn't handle a case where one point in a timeseries needed to be labelled with an 'erroneous' label. In such a case the entire timeseries would have to be labelled 'erroneous' (or a new timeseries made with just that data point, which may not be very performant). If behaviour of this type is required, we suggest trying another data handling approach.

## The ScmDatabase class

When handling large datasets which may not fit into memory, it is important to be able to query subsets of the dataset without having to iterate over the entire dataset. `scmdata.database.ScmDatabase` helps with this issue by disaggregating a dataset into subsets according to unique combinations of metadata. The metadata of interest is specified by the user so that the database can be adapted to any use-case or access pattern.

One of the major benefits of `scmdata.database.ScmDatabase` is that the taxonomy of metadata does not need to be known at database creation making it easy to add new data to the database. Each unique subset of the database is stored as a single netCDF file. This ensures that if timeseries with new metadata are saved to the database, the existing files in the database do not need to be modified. Instead new files are written expanding the directory structure to accommodate the new metadata values.

Filtering using the metadata columns of interest is also very simple as the contents of a given file can be determined from the directory structure without having to load the file. Each file can then be loaded as the data is needed, minimising the need for reading data which will then immediately be filtered away of extra data that is needed to be unnecessarily read and then filtered away.

### 1.1.3 Development

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

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

## Contributing

All contributions are welcome, some possible suggestions include:

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

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

As a contributor, please follow a couple of conventions:

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

## Getting setup

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

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



## Getting help

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

## Development tools

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

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

## Other tools

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

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

## Formatting

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

### Buiding the docs

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

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

### Gotchas

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

### Docstring style

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

### Releasing

#### First step

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

### PyPI

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

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

Assuming test PyPI worked, now upload to the main repository

1. `make publish-on-pypi`
2. Go to [SCMData’s PyPI](#) and check that the new release is as intended.

3. Test the install with `make test-pypi-install`

## Conda

1. If you haven't already, fork the [SCMData conda feedstock](#). In your fork, add the feedstock upstream with `git remote add upstream https://github.com/conda-forge/scmdata-feedstock` (upstream should now appear in the output of `git remote -v`)
2. Update your fork's master to the upstream master with:
  1. `git checkout master`
  2. `git fetch upstream`
  3. `git reset --hard upstream/master`
3. Create a new branch in the feedstock for the version you want to bump to.
4. Edit `recipe/meta.yaml` and update:
  - version number in line 2 (don't include the 'v' in the version tag)
  - the build number to zero in line 13 (you should only be here if releasing a new version)
  - update sha256 in line 10 (you can get the sha from [SCMData's PyPI](#) by clicking on 'Download files' on the left and then clicking on 'SHA256' of the `.tar.gz` file to copy it to the clipboard)
5. `git add .`
6. `git commit -m "Update to vX.Y.Z"`
7. `git push`
8. Make a PR into the [SCMData conda feedstock](#)
9. If the PR passes (give it at least 10 minutes to run all the CI), merge
10. Check <https://anaconda.org/conda-forge/scmdata> to double check that the version has increased (this can take a few minutes to update)

## Push to repository

Finally, push the tags and commit to the repository

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

## Why is there a Makefile in a pure Python repository?

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

### 1.1.4 scmdata.database

Database for handling large datasets in a performant, but flexible way

Data is chunked using unique combinations of metadata. This allows for the database to expand as new data is added without having to change any of the existing data.

Subsets of data are also able to be read without having to load all the data and then filter. For example, one could save model results from a number of different climate models and then load just the Surface Temperature data for all models.

**class** scmdata.database.DatabaseBackend(\*\*kwargs)

Bases: `abc.ABC`

Abstract backend for serialising/deserialising data

Data is stored as objects represented by keys. These keys can be used later to load data.

**delete**(key)

Delete a given key

**Parameters** **key** (`str`) –

**abstract get**(filters)

Get all matching keys for a given filter

**Parameters** **filters** (*dict of str*) – String filters If a level is missing then all values are fetched

**Returns** Each item is a key which may contain data which is of interest

**Return type** list of str

**abstract load**(key)

Load data at a given key

**Parameters** **key** (`str`) – Key to load

**Returns**

**Return type** scmdata.ScmRun

**abstract save**(sr)

Save data

**Parameters** **sr** (`scmdata.ScmRun`) –

**Returns** Key where the data is stored

**Return type** `str`

**class** scmdata.database.NetCDFBackend(\*\*kwargs)

Bases: `scmdata.database.DatabaseBackend`

On-disk database handler for outputs from SCMs

Data is split into groups as specified by `levels`. This allows for fast reading and writing of new subsets of data when a single output file is no longer performant or data cannot all fit in memory.

**delete**(key)

Delete a key

**Parameters** **key** (`str`) –

**get**(filters)

Get all matching objects for a given filter

**Parameters** **filters** (*dict of str*) – String filters If a level is missing then all values are fetched

**Returns**

**Return type** list of str

**get\_key**(*sr*)

Get key where the data will be stored

The key is the root directory joined with the other information provided. The filepath is also cleaned to remove spaces and special characters.

**Parameters** **sr** (`scmdata.ScmRun`) – Data to save

**Raises**

- **ValueError** – If non-unique metadata is found for each of `self.kwargs["levels"]`
- **KeyError** – If missing metadata is found for each of `self.kwargs["levels"]`

**Returns** Path in which to save the data without spaces or special characters

**Return type** str

**load**(*key*)

**Parameters** **key** (*str*) –

**Returns**

**Return type** `scmdata.ScmRun`

**save**(*sr*)

Save a ScmRun to the database

The dataset should not contain any duplicate metadata for the database levels

**Parameters** **sr** (`scmdata.ScmRun`) – Data to save

**Raises**

- **ValueError** – If duplicate metadata are present for the requested database levels
- **KeyError** – If metadata for the requested database levels are not found

**Returns** Key where the data is saved

**Return type** str

**class** `scmdata.database.ScmDatabase`(*root\_dir*, *levels*=(*'climate\_model'*, *'variable'*, *'region'*, *'scenario'*), *backend*=*'netcdf'*, *backend\_config*=*None*)

Bases: `object`

On-disk database handler for outputs from SCMs

Data is split into groups as specified by `levels`. This allows for fast reading and writing of new subsets of data when a single output file is no longer performant or data cannot all fit in memory.

**available\_data**()

Get all the data which is available to be loaded

If metadata includes non-alphanumeric characters then it might appear modified in the returned table. The original metadata values can still be used to filter data.

**Returns**

**Return type** `pd.DataFrame`

**delete**(*\*\*filters*)

Delete data from the database

**Parameters** **filters** (*dict of str*) – Filters for the data to load.

Defaults to deleting all data if nothing is specified.

**Raises** **ValueError** – If a filter for a level not in `levels` is specified

**load**(*disable\_tqdm=False, \*\*filters*)

Load data from the database

**Parameters**

- **disable\_tqdm** (*bool*) – If True, do not show the progress bar
- **filters** (*dict of str : [str, list[str]]*) – Filters for the data to load.

Defaults to loading all values for a level if it isn't specified.

If a filter is a list then OR logic is applied within the level. For example, if we have `scenario=["ssp119", "ssp126"]` then both the ssp119 and ssp126 scenarios will be loaded.

**Returns** Loaded data

**Return type** `scmdata.ScmRun`

**Raises** **ValueError** – If a filter for a level not in `levels` is specified

If no data matching `filters` is found

**property** `root_dir`

Root directory of the database.

**Returns**

**Return type** `str`

**save**(*scmrun, disable\_tqdm=False*)

Save data to the database

The results are saved with one file for each unique combination of `levels` in a directory structure underneath `root_dir`.

Use `available_data()` to see what data is available. Subsets of data can then be loaded as an `scmdata.ScmRun` using `load()`.

**Parameters**

- **scmrun** (`scmdata.ScmRun`) – Data to save.  
The timeseries in this run should have valid metadata for each of the columns specified in `levels`.
- **disable\_tqdm** (*bool*) – If True, do not show the progress bar

**Raises** **KeyError** – If a filter for a level not in `levels` is specified

`scmdata.database.ensure_dir_exists(fp)`

Ensure directory exists

**Parameters** **fp** (*str*) – Filepath of which to ensure the directory exists

### 1.1.5 scmdata.errors

Custom errors and exceptions used by scmdata

**exception** `scmdata.errors.DuplicateTimesError(time_index)`

Bases: `ValueError`

Error raised when times are duplicated

**exception** `scmdata.errors.MissingRequiredColumnError(columns)`

Bases: `ValueError`

Error raised when an operation produces missing metadata columns

**exception** `scmdata.errors.NonUniqueMetadataError(meta)`

Bases: `ValueError`

Error raised when metadata is not unique

### 1.1.6 scmdata.filters

Helpers for filtering data in `scmdata.run.ScmRun`.

Based upon `pyam.utils`.

`scmdata.filters.datetime_match(data: List, dts: Union[List[datetime.datetime], datetime.datetime]) → numpy.ndarray`

Match datetimes in time columns for data filtering.

#### Parameters

- **data** – Input data to perform filtering on
- **dts** – Datetimes to use for filtering

**Returns** Array where True indicates a match

**Return type** `numpy.ndarray` of `bool`

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

`scmdata.filters.day_match(data: List, days: Union[List[str], List[int], int, str]) → numpy.ndarray`

Match days in time columns for data filtering.

#### Parameters

- **data** – Input data to perform filtering on
- **days** – Days to match

**Returns** Array where True indicates a match

**Return type** `numpy.ndarray` of `bool`

`scmdata.filters.find_depth(meta_col: pandas.core.series.Series, s: str, level: Union[int, str], separator: str = '|') → numpy.ndarray`

Find all values which match given depth from a filter keyword.

#### Parameters

- **meta\_col** – Column in which to find values which match the given depth
- **s** – Filter keyword, from which level should be applied

- **level** – 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** – The string used to separate levels in s. Defaults to a pipe (“|”).

**Returns** Array where True indicates a match

**Return type** `numpy.ndarray` of `bool`

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

`scmdata.filters.hour_match(data: List, hours: Union[List[int], int]) → numpy.ndarray`

Match hours in time columns for data filtering.

**Parameters**

- **data** – Input data to perform filtering on
- **hours** – Hours to match

**Returns** Array where True indicates a match

**Return type** `numpy.ndarray` of `bool`

`scmdata.filters.is_in(vals: List, items: List) → numpy.ndarray`

Find elements of vals which are in items.

**Parameters**

- **vals** – The list of values to check
- **items** – 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** `numpy.ndarray` of `bool`

`scmdata.filters.month_match(data: List, months: Union[List[str], List[int], int, str]) → numpy.ndarray`

Match months in time columns for data filtering.

**Parameters**

- **data** – Input data to perform filtering on
- **months** – Months to match

**Returns** Array where True indicates a match

**Return type** `numpy.ndarray` of `bool`

`scmdata.filters.pattern_match(meta_col: pandas.core.series.Series, values: Union[Iterable[str], str], level: Optional[Union[str, int]] = None, regexp: bool = False, separator: str = '|') → numpy.ndarray`

Filter data by matching metadata columns to given patterns.

**Parameters**

- **meta\_col** – Column to perform filtering on
- **values** – Values to match
- **level** – Passed to `find_depth()`. For usage, see docstring of `find_depth()`.
- **regexp** – If `True`, match using regexp rather than our pseudo regexp syntax.



- **has\_nan** – If True, convert all nan values in meta\_col to empty string before applying filters. This means that “” and “\*” will match rows with `numpy.nan`. If False, the conversion is not applied and so a search in a string column which contains `numpy.nan` will result in a `TypeError`.
- **separator** – String used to separate the hierarchy levels in values. Defaults to ‘|’

**Returns** Array where True indicates a match

**Return type** `numpy.ndarray` of `bool`

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

`scmdata.filters.time_match(data: List, times: Union[List[str], List[int], int, str], conv_codes: List[str], strptime_attr: str, name: str) → numpy.ndarray`

Match times by applying conversion codes to filtering list.

#### Parameters

- **data** – Input data to perform filtering on
- **times** – Times to match
- **conv\_codes** – If times contains strings, conversion codes to try passing to `time.strptime()` to convert times to `datetime.datetime`
- **strptime\_attr** – If times contains strings, the `datetime.datetime` attribute to finalize the conversion of strings to integers
- **name** – Name of the part of a datetime to extract, used to produce useful error messages.

**Returns** Array where True indicates a match

**Return type** `numpy.ndarray` 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)

`scmdata.filters.years_match(data: List, years: Union[List[int], numpy.ndarray, int]) → numpy.ndarray`

Match years in time columns for data filtering.

#### Parameters

- **data** – Input data to perform filtering on
- **years** – Years to match

**Returns** Array where True indicates a match

**Return type** `numpy.ndarray` of `bool`

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

### 1.1.7 scmdata.groupby

Functionality for grouping and filtering ScmRun objects

**class** scmdata.groupby.**RunGroupBy**(*run, groups*)

Bases: scmdata.groupby.\_GroupBy

GroupBy object specialized to grouping ScmRun objects

**all**(*dim=None, axis=None, \*\*kwargs*)

Reduce this RunGroupBy's data by applying *all* along some dimension(s).

#### Parameters

- **dim**(*str* or *sequence of str, optional*) – Dimension(s) over which to apply *all*.
- **axis**(*int* or *sequence of int, optional*) – Axis(es) over which to apply *all*. Only one of the 'dim' and 'axis' arguments can be supplied. If neither are supplied, then *all* is calculated over axes.
- **keep\_attrs**(*bool, optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- **\*\*kwargs**(*dict*) – Additional keyword arguments passed on to the appropriate array function for calculating *all* on this object's data.

**Returns** **reduced** – New RunGroupBy object with *all* applied to its data and the indicated dimension(s) removed.

**Return type** *RunGroupBy*

**any**(*dim=None, axis=None, \*\*kwargs*)

Reduce this RunGroupBy's data by applying *any* along some dimension(s).

#### Parameters

- **dim**(*str* or *sequence of str, optional*) – Dimension(s) over which to apply *any*.
- **axis**(*int* or *sequence of int, optional*) – Axis(es) over which to apply *any*. Only one of the 'dim' and 'axis' arguments can be supplied. If neither are supplied, then *any* is calculated over axes.
- **keep\_attrs**(*bool, optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- **\*\*kwargs**(*dict*) – Additional keyword arguments passed on to the appropriate array function for calculating *any* on this object's data.

**Returns** **reduced** – New RunGroupBy object with *any* applied to its data and the indicated dimension(s) removed.

**Return type** *RunGroupBy*

**count**(*dim=None, axis=None, \*\*kwargs*)

Reduce this RunGroupBy's data by applying *count* along some dimension(s).

#### Parameters

- **dim**(*str* or *sequence of str, optional*) – Dimension(s) over which to apply *count*.

- **axis** (*int* or *sequence of int*, *optional*) – Axis(es) over which to apply *count*. Only one of the ‘dim’ and ‘axis’ arguments can be supplied. If neither are supplied, then *count* is calculated over axes.
- **keep\_attrs** (*bool*, *optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- **\*\*kwargs** (*dict*) – Additional keyword arguments passed on to the appropriate array function for calculating *count* on this object’s data.

**Returns** **reduced** – New RunGroupBy object with *count* applied to its data and the indicated dimension(s) removed.

**Return type** *RunGroupBy*

**map**(*func*, *\*args*, *\*\*kwargs*)

Apply a function to each group and append the results

*func* is called like *func(ar, \*args, \*\*kwargs)* for each *ScmRun* *ar* in this group. If the result of this function call is None, then it is excluded from the results.

The results are appended together using *run\_append()*. The function can change the size of the input *ScmRun* as long as *run\_append()* can be applied to all results.

## Examples

```
>>> def write_csv(arr):
...     variable = arr.get_unique_meta("variable")
...     arr.to_csv("out-{}.csv".format(variable))
>>> df.groupby("variable").map(write_csv)
```

### Parameters

- **func** (*function*) – Callable to apply to each timeseries.
- **\*args** – Positional arguments passed to *func*.
- **\*\*kwargs** – Used to call *func(ar, \*\*kwargs)* for each array *ar*.

**Returns** **applied** – The result of splitting, applying and combining this array.

**Return type** *ScmRun*

**max**(*dim=None*, *axis=None*, *skipna=None*, *\*\*kwargs*)

Reduce this RunGroupBy’s data by applying *max* along some dimension(s).

### Parameters

- **dim** (*str* or *sequence of str*, *optional*) – Dimension(s) over which to apply *max*.
- **axis** (*int* or *sequence of int*, *optional*) – Axis(es) over which to apply *max*. Only one of the ‘dim’ and ‘axis’ arguments can be supplied. If neither are supplied, then *max* is calculated over axes.
- **skipna** (*bool*, *optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).

- **keep\_attrs** (*bool*, *optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- **\*\*kwargs** (*dict*) – Additional keyword arguments passed on to the appropriate array function for calculating *max* on this object's data.

**Returns** **reduced** – New RunGroupBy object with *max* applied to its data and the indicated dimension(s) removed.

**Return type** *RunGroupBy*

**mean**(*dim=None*, *axis=None*, *skipna=None*, *\*\*kwargs*)

Reduce this RunGroupBy's data by applying *mean* along some dimension(s).

**Parameters**

- **dim** (*str* or *sequence of str*, *optional*) – Dimension(s) over which to apply *mean*.
- **axis** (*int* or *sequence of int*, *optional*) – Axis(es) over which to apply *mean*. Only one of the 'dim' and 'axis' arguments can be supplied. If neither are supplied, then *mean* is calculated over axes.
- **skipna** (*bool*, *optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **keep\_attrs** (*bool*, *optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- **\*\*kwargs** (*dict*) – Additional keyword arguments passed on to the appropriate array function for calculating *mean* on this object's data.

**Returns** **reduced** – New RunGroupBy object with *mean* applied to its data and the indicated dimension(s) removed.

**Return type** *RunGroupBy*

**median**(*dim=None*, *axis=None*, *skipna=None*, *\*\*kwargs*)

Reduce this RunGroupBy's data by applying *median* along some dimension(s).

**Parameters**

- **dim** (*str* or *sequence of str*, *optional*) – Dimension(s) over which to apply *median*.
- **axis** (*int* or *sequence of int*, *optional*) – Axis(es) over which to apply *median*. Only one of the 'dim' and 'axis' arguments can be supplied. If neither are supplied, then *median* is calculated over axes.
- **skipna** (*bool*, *optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **keep\_attrs** (*bool*, *optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- **\*\*kwargs** (*dict*) – Additional keyword arguments passed on to the appropriate array function for calculating *median* on this object's data.

**Returns reduced** – New RunGroupBy object with *median* applied to its data and the indicated dimension(s) removed.

**Return type** *RunGroupBy*

**min**(*dim=None, axis=None, skipna=None, \*\*kwargs*)

Reduce this RunGroupBy's data by applying *min* along some dimension(s).

#### Parameters

- **dim**(*str* or *sequence of str, optional*) – Dimension(s) over which to apply *min*.
- **axis**(*int* or *sequence of int, optional*) – Axis(es) over which to apply *min*. Only one of the 'dim' and 'axis' arguments can be supplied. If neither are supplied, then *min* is calculated over axes.
- **skipna**(*bool, optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **keep\_attrs**(*bool, optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- **\*\*kwargs**(*dict*) – Additional keyword arguments passed on to the appropriate array function for calculating *min* on this object's data.

**Returns reduced** – New RunGroupBy object with *min* applied to its data and the indicated dimension(s) removed.

**Return type** *RunGroupBy*

**prod**(*dim=None, axis=None, skipna=None, \*\*kwargs*)

Reduce this RunGroupBy's data by applying *prod* along some dimension(s).

#### Parameters

- **dim**(*str* or *sequence of str, optional*) – Dimension(s) over which to apply *prod*.
- **axis**(*int* or *sequence of int, optional*) – Axis(es) over which to apply *prod*. Only one of the 'dim' and 'axis' arguments can be supplied. If neither are supplied, then *prod* is calculated over axes.
- **skipna**(*bool, optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **min\_count**(*int, default: None*) – The required number of valid values to perform the operation. If fewer than min\_count non-NA values are present the result will be NA. Only used if skipna is set to True or defaults to True for the array's dtype. New in version 0.10.8: Added with the default being None. Changed in version 0.17.0: if specified on an integer array and skipna=True, the result will be a float array.
- **keep\_attrs**(*bool, optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- **\*\*kwargs**(*dict*) – Additional keyword arguments passed on to the appropriate array function for calculating *prod* on this object's data.

**Returns reduced** – New RunGroupBy object with *prod* applied to its data and the indicated dimension(s) removed.

**Return type** *RunGroupBy*

**reduce**(*func*, *dim=None*, *axis=None*, *\*\*kwargs*)

Reduce the items in this group by applying *func* along some dimension(s).

**Parameters**

- **func** (*function*) – Function which can be called in the form *func(x, axis=axis, \*\*kwargs)* to return the result of collapsing an np.ndarray over an integer valued axis.
- **dim** (*...*, *str* or *sequence of str*, *optional*) – Not used in this implementation
- **axis** (*int* or *sequence of int*, *optional*) – Axis(es) over which to apply *func*. Only one of the ‘dimension’ and ‘axis’ arguments can be supplied. If neither are supplied, then *func* is calculated over all dimension for each group item.
- **\*\*kwargs** (*dict*) – Additional keyword arguments passed on to *func*.

**Returns** **reduced** – Array with summarized data and the indicated dimension(s) removed.

**Return type** *ScmRun*

**std**(*dim=None*, *axis=None*, *skipna=None*, *\*\*kwargs*)

Reduce this RunGroupBy’s data by applying *std* along some dimension(s).

**Parameters**

- **dim** (*str* or *sequence of str*, *optional*) – Dimension(s) over which to apply *std*.
- **axis** (*int* or *sequence of int*, *optional*) – Axis(es) over which to apply *std*. Only one of the ‘dim’ and ‘axis’ arguments can be supplied. If neither are supplied, then *std* is calculated over axes.
- **skipna** (*bool*, *optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **keep\_attrs** (*bool*, *optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- **\*\*kwargs** (*dict*) – Additional keyword arguments passed on to the appropriate array function for calculating *std* on this object’s data.

**Returns** **reduced** – New RunGroupBy object with *std* applied to its data and the indicated dimension(s) removed.

**Return type** *RunGroupBy*

**sum**(*dim=None*, *axis=None*, *skipna=None*, *\*\*kwargs*)

Reduce this RunGroupBy’s data by applying *sum* along some dimension(s).

**Parameters**

- **dim** (*str* or *sequence of str*, *optional*) – Dimension(s) over which to apply *sum*.
- **axis** (*int* or *sequence of int*, *optional*) – Axis(es) over which to apply *sum*. Only one of the ‘dim’ and ‘axis’ arguments can be supplied. If neither are supplied, then *sum* is calculated over axes.
- **skipna** (*bool*, *optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **min\_count** (*int*, *default: None*) – The required number of valid values to perform the operation. If fewer than min\_count non-NA values are present the result will be NA. Only used if skipna is set to True or defaults to True for the array’s dtype. New in version

0.10.8: Added with the default being None. Changed in version 0.17.0: if specified on an integer array and skipna=True, the result will be a float array.

- **keep\_attrs** (*bool*, *optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- **\*\*kwargs** (*dict*) – Additional keyword arguments passed on to the appropriate array function for calculating *sum* on this object's data.

**Returns** **reduced** – New RunGroupBy object with *sum* applied to its data and the indicated dimension(s) removed.

**Return type** *RunGroupBy*

**var**(*dim=None*, *axis=None*, *skipna=None*, **\*\*kwargs**)

Reduce this RunGroupBy's data by applying *var* along some dimension(s).

**Parameters**

- **dim** (*str* or *sequence of str*, *optional*) – Dimension(s) over which to apply *var*.
- **axis** (*int* or *sequence of int*, *optional*) – Axis(es) over which to apply *var*. Only one of the 'dim' and 'axis' arguments can be supplied. If neither are supplied, then *var* is calculated over axes.
- **skipna** (*bool*, *optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **keep\_attrs** (*bool*, *optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- **\*\*kwargs** (*dict*) – Additional keyword arguments passed on to the appropriate array function for calculating *var* on this object's data.

**Returns** **reduced** – New RunGroupBy object with *var* applied to its data and the indicated dimension(s) removed.

**Return type** *RunGroupBy*

## 1.1.8 scmdata.netcdf

NetCDF4 file operations

Reading and writing *ScmRun* to disk as binary

`scmdata.netcdf.inject_nc_methods(cls)`

Add the to/from nc methods to a class

**Parameters** **cls** – Class to add methods to

`scmdata.netcdf.nc_to_run(cls, fname)`

Read a netCDF4 file from disk

**Parameters** **fname** (*str*) – Filename to read

**See also:**

`scmdata.run.ScmRun.from_nc()`



```
scmdata.netcdf.run_to_nc(run, fname, dimensions=('region',), extras=(), **kwargs)
```

Write timeseries to disk as a netCDF4 file

Each unique variable will be written as a variable within the netCDF file. Choosing the dimensions and extras such that there are as few empty (or nan) values as possible will lead to the best compression on disk.

#### Parameters

- **fname** (*str*) – Path to write the file into
- **dimensions** (*iterable of str*) – Dimensions to include in the netCDF file. The time dimension is always included (if not provided it will be the last dimension). An additional dimension (specifically a co-ordinate in xarray terms), “\_id”, will be included if **extras** is provided and any of the metadata in **extras** is not uniquely defined by **dimensions**. “\_id” maps the timeseries in each variable to their relevant metadata.
- **extras** (*iterable of str*) – Metadata columns to write as variables in the netCDF file (specifically as “non-dimension co-ordinates” in xarray terms, see [xarray terminology](#) for more details). Where possible, these non-dimension co-ordinates will use dimension co-ordinates as their own co-ordinates. However, if the metadata in **extras** is not defined by a single dimension in **dimensions**, then the **extras** co-ordinates will have dimensions of “\_id”. This “\_id” co-ordinate maps the values in the **extras** co-ordinates to each timeseries in the serialised dataset. Where “\_id” is required, an extra “\_id” dimension will also be added to **dimensions**.
- **kwargs** – Passed through to `xarray.Dataset.to_netcdf()`

See also:

`scmdata.run.ScmRun.to_nc()`

### 1.1.9 scmdata.offsets

Allow stepping through time using xarray’s offset functionality

Provides similar functionality to [https://pandas.pydata.org/pandas-docs/stable/user\\_guide/timeseries.html#dateoffset-objects](https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#dateoffset-objects)

```
scmdata.offsets.generate_range(start: cftime._cftime.datetime, end: cftime._cftime.datetime, offset:
                               xarray.coding.cftime_offsets.BaseCFTimeOffset, date_cls:
                               cftime._cftime.datetime = <class 'cftime._cftime.DatetimeGregorian'>) →
                               Iterable[cftime._cftime.datetime]
```

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** (*cftime.datetime*) – Starting datetime from which to generate the range (noting roll backward mentioned above and illustrated in the examples).
- **end** (*cftime.datetime*) – Last datetime from which to generate the range (noting roll forward mentioned above and illustrated in the examples).
- **offset** – Offset object for determining the timesteps.
- **date\_cls** (*cftime.datetime*) – The time points will be returned as instances of `date_cls`

**Yields** *cftime.datetime* – Next datetime in the range (the exact class is specified by `date_cls`)

**Raises** `ValueError` – Offset does not result in increasing *cftime.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 scmdata.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])
[cftime.DatetimeGregorian(2001, 1, 1, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2002, 1, 1, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2003, 1, 1, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2004, 1, 1, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2005, 1, 1, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2006, 1, 1, 0, 0, 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])
[cftime.DatetimeGregorian(2000, 12, 31, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2001, 12, 31, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2002, 12, 31, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2003, 12, 31, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2004, 12, 31, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2005, 12, 31, 0, 0, 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])
[cftime.DatetimeGregorian(2001, 4, 1, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2001, 7, 1, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2001, 10, 1, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2002, 1, 1, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2002, 4, 1, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2002, 7, 1, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2002, 10, 1, 0, 0, 0, 0),
 cftime.DatetimeGregorian(2003, 1, 1, 0, 0, 0, 0),
```

(continues on next page)

(continued from previous page)

```

cftime.DatetimeGregorian(2003, 4, 1, 0, 0, 0, 0),
cftime.DatetimeGregorian(2003, 7, 1, 0, 0, 0, 0),
cftime.DatetimeGregorian(2003, 10, 1, 0, 0, 0, 0),
cftime.DatetimeGregorian(2004, 1, 1, 0, 0, 0, 0),
cftime.DatetimeGregorian(2004, 4, 1, 0, 0, 0, 0),
cftime.DatetimeGregorian(2004, 7, 1, 0, 0, 0, 0),
cftime.DatetimeGregorian(2004, 10, 1, 0, 0, 0, 0),
cftime.DatetimeGregorian(2005, 1, 1, 0, 0, 0, 0),
cftime.DatetimeGregorian(2005, 4, 1, 0, 0, 0, 0),
cftime.DatetimeGregorian(2005, 7, 1, 0, 0, 0, 0)]

```

### 1.1.10 scmdata.ops

Operations for *ScmRun* objects

These largely rely on Pint’s Pandas interface to handle unit conversions automatically

`scmdata.ops.add(self, other, op_cols, **kwargs)`

Add values

#### Parameters

- **other** (*ScmRun*) – *ScmRun* containing data to add
- **op\_cols** (*dict of str: str*) – Dictionary containing the columns to drop before adding as the keys and the value those columns should hold in the output as the values. For example, if we have `op_cols={"variable": "Emissions|CO2 - Emissions|CO2|Fossil"}` then the addition will be performed with an index that uses all columns except the “variable” column and the output will have a “variable” column with the value “Emissions|CO2 - Emissions|CO2|Fossil”.
- **\*\*kwargs** (*any*) – Passed to `prep_for_op()`

**Returns** Sum of `self` and `other`, using `op_cols` to define the columns which should be dropped before the data is aligned and to define the value of these columns in the output.

**Return type** *ScmRun*

#### Examples

```

>>> import numpy as np
>>> from scmdata import ScmRun
>>>
>>> IDX = [2010, 2020, 2030]
>>>
>>> start = ScmRun(
...     data=np.arange(18).reshape(3, 6),
...     index=IDX,
...     columns={
...         "variable": [
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Emissions|CO2|Fossil",

```

(continues on next page)

(continued from previous page)

```

...         "Emissions|CO2|AFOLU",
...         "Cumulative Emissions|CO2",
...         "Surface Air Temperature Change",
...     ],
...     "unit": ["GtC / yr", "GtC / yr", "GtC / yr", "GtC / yr", "GtC", "K"],
...     "region": ["World|NH", "World|NH", "World|SH", "World|SH", "World",
→ "World"],
...     "model": "idealised",
...     "scenario": "idealised",
... },
... )
>>>
>>> start.head()
time                                     2010-01-01 00:00:00_
→ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|Fossil  GtC / yr World|NH idealised idealised      0.0_
→          6.0              12.0
Emissions|CO2|AFOLU   GtC / yr World|NH idealised idealised      1.0_
→          7.0              13.0
Emissions|CO2|Fossil  GtC / yr World|SH idealised idealised      2.0_
→          8.0              14.0
Emissions|CO2|AFOLU   GtC / yr World|SH idealised idealised      3.0_
→          9.0              15.0
Cumulative Emissions|CO2 GtC      World  idealised idealised      4.0_
→          10.0             16.0
>>> fos = start.filter(variable="*Fossil")
>>> fos.head()
time                                     2010-01-01 00:00:00 _
→ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|Fossil GtC / yr World|NH idealised idealised      0.0  _
→          6.0              12.0
→          World|SH idealised idealised      2.0  _
→          8.0              14.0
>>>
>>> afolu = start.filter(variable="*AFOLU")
>>> afolu.head()
time                                     2010-01-01 00:00:00 _
→ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|AFOLU GtC / yr World|NH idealised idealised      1.0  _
→          7.0              13.0
→          World|SH idealised idealised      3.0  _
→          9.0              15.0
>>>
>>> total = fos.add(afolu, op_cols={"variable": "Emissions|CO2"})
>>> total.head()
time                                     2010-01-01 00:00:00  2020-01-
→ 01 00:00:00  2030-01-01 00:00:00
model      scenario region  variable      unit
idealised idealised World|NH Emissions|CO2 gigatC / a      1.0  _
→          13.0              25.0

```

(continues on next page)

(continued from previous page)

```

World|SH Emissions|CO2 gigatC / a          5.0
↪ 17.0          29.0
>>>
>>> nh = start.filter(region="*NH")
>>> nh.head()
time                                     2010-01-01 00:00:00
↪ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|Fossil GtC / yr World|NH idealised idealised      0.0
↪          6.0          12.0
Emissions|CO2|AFOLU  GtC / yr World|NH idealised idealised      1.0
↪          7.0          13.0
>>>
>>> sh = start.filter(region="*SH")
>>> sh.head()
time                                     2010-01-01 00:00:00
↪ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|Fossil GtC / yr World|SH idealised idealised      2.0
↪          8.0          14.0
Emissions|CO2|AFOLU  GtC / yr World|SH idealised idealised      3.0
↪          9.0          15.0
>>>
>>> world = nh.add(sh, op_cols={"region": "World"})
>>> world.head()
time                                     2010-01-01 00:00:00
↪ 2020-01-01 00:00:00  2030-01-01 00:00:00
model      scenario region variable      unit
idealised idealised World  Emissions|CO2|Fossil gigatC / a      2.0
↪          14.0          26.0
          Emissions|CO2|AFOLU  gigatC / a      4.0
↪          16.0          28.0

```

```

scmdata.ops.adjust_median_to_target(self, target, evaluation_period, process_over=None,
                                     check_groups_identical=False,
                                     check_groups_identical_kwargs=None)

```

Adjust the median of (an ensemble of) timeseries to a specified target

#### Parameters

- **target** (*float*) – Value to which the median of each (group of) timeseries should be adjusted
- **evaluation\_period** (*list[int]*) – Period over which the median should be evaluated
- **process\_over** (*list*) – Metadata to treat as ‘ensemble members’ i.e. all other columns in the metadata of *self* will be used to group the timeseries before calculating the median. If not supplied, timeseries will not be grouped.
- **check\_groups\_identical** (*bool*) – Should we check that the median of each group is the same before making the adjustment?
- **check\_groups\_identical\_kwargs** (*dict*) – Only used if *check\_groups\_identical* is True, in which case these are passed through to *np.testing.assert\_allclose*

#### Raises

- **NotImplementedError** – `evaluation_period` is based on times not years
- **AssertionError** – If `check_groups_identical` is `True` and the median of each group is not the same before making the adjustment.

**Returns** Timeseries adjusted to have the intended median

**Return type** `ScmRun`

`scmdata.ops.delta_per_delta_time(self, out_var=None)`

Calculate change in timeseries values for each timestep, divided by the size of the timestep

The output is placed on the middle of each timestep and is one timestep shorter than the input.

**Parameters** `out_var` (`str`) – If provided, the variable column of the output is set equal to `out_var`. Otherwise, the output variables are equal to the input variables, prefixed with “Delta ”.

**Returns** `scmdata.ScmRun` containing the changes in values of `self`, normalised by the change in time

**Return type** `scmdata.ScmRun`

**Warns** **UserWarning** – The data contains nans. If this happens, the output data will also contain nans.

`scmdata.ops.divide(self, other, op_cols, **kwargs)`

Divide values (self / other)

**Parameters**

- **other** (`ScmRun`) – `ScmRun` containing data to divide
- **op\_cols** (`dict of str: str`) – Dictionary containing the columns to drop before dividing as the keys and the value those columns should hold in the output as the values. For example, if we have `op_cols={"variable": "Emissions|CO2 - Emissions|CO2|Fossil"}` then the division will be performed with an index that uses all columns except the “variable” column and the output will have a “variable” column with the value “Emissions|CO2 - Emissions|CO2|Fossil”.
- **\*\*kwargs** (`any`) – Passed to `prep_for_op()`

**Returns** Quotient of `self` and `other`, using `op_cols` to define the columns which should be dropped before the data is aligned and to define the value of these columns in the output.

**Return type** `ScmRun`

## Examples

```
>>> import numpy as np
>>> from scmdata import ScmRun
>>>
>>> IDX = [2010, 2020, 2030]
>>>
>>>
>>> start = ScmRun(
...     data=np.arange(18).reshape(3, 6),
...     index=IDX,
...     columns={
...         "variable": [
...             "Emissions|CO2|Fossil",
```

(continues on next page)

(continued from previous page)

```

...         "Emissions|CO2|AFOLU",
...         "Emissions|CO2|Fossil",
...         "Emissions|CO2|AFOLU",
...         "Cumulative Emissions|CO2",
...         "Surface Air Temperature Change",
...     ],
...     "unit": ["GtC / yr", "GtC / yr", "GtC / yr", "GtC / yr", "GtC", "K"],
...     "region": ["World|NH", "World|NH", "World|SH", "World|SH", "World",
→ "World"],
...     "model": "idealised",
...     "scenario": "idealised",
... },
... )
>>>
>>> start.head()
time                                     2010-01-01 00:00:00
→ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|Fossil  GtC / yr World|NH idealised idealised      0.0
→          6.0              12.0
Emissions|CO2|AFOLU   GtC / yr World|NH idealised idealised      1.0
→          7.0              13.0
Emissions|CO2|Fossil  GtC / yr World|SH idealised idealised      2.0
→          8.0              14.0
Emissions|CO2|AFOLU   GtC / yr World|SH idealised idealised      3.0
→          9.0              15.0
Cumulative Emissions|CO2 GtC      World  idealised idealised      4.0
→          10.0             16.0
>>> fos = start.filter(variable="*Fossil")
>>> fos.head()
time                                     2010-01-01 00:00:00
→ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|Fossil GtC / yr World|NH idealised idealised      0.0
→          6.0              12.0
→          8.0              14.0
World|SH idealised idealised      2.0
→          14.0
>>>
>>> afolu = start.filter(variable="*AFOLU")
>>> afolu.head()
time                                     2010-01-01 00:00:00
→ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|AFOLU GtC / yr World|NH idealised idealised      1.0
→          7.0              13.0
→          9.0              15.0
World|SH idealised idealised      3.0
→          15.0
>>>
>>> fos_afolu_ratio = fos.divide(
...     afolu, op_cols={"variable": "Emissions|CO2|Fossil : AFOLU"}
... )
>>> fos_afolu_ratio.head()

```

(continues on next page)

(continued from previous page)

time				2010-01-01
→00:00:00	2020-01-01 00:00:00	2030-01-01 00:00:00		
model	scenario	region	variable	unit
idealised	idealised	World NH	Emissions CO2 Fossil : AFOLU	dimensionless
→0.000000		0.857143	0.923077	
		World SH	Emissions CO2 Fossil : AFOLU	dimensionless
→0.666667		0.888889	0.933333	

`scmdata.ops.inject_ops_methods(cls)`

Inject the operation methods

**Parameters** `cls` – Target class

`scmdata.ops.integrate(self, out_var=None)`

Integrate with respect to time

**Parameters** `out_var` (`str`) – If provided, the variable column of the output is set equal to `out_var`. Otherwise, the output variables are equal to the input variables, prefixed with “Cumulative”.

**Returns** `scmdata.ScmRun` containing the integral of `self` with respect to time

**Return type** `scmdata.ScmRun`

**Warns** `UserWarning` – The data being integrated contains nans. If this happens, the output data will also contain nans.

`scmdata.ops.linear_regression(self)`

Calculate linear regression of each timeseries

---

**Note:** Times in seconds since 1970-01-01 are used as the x-axis for the regressions. Such values can be accessed with `self.time_points.values.astype("datetime64[s]").astype("int")`. This decision does not matter for the gradients, but is important for the intercept values.

---

**Returns** list of dict[`str`] – List of dictionaries. Each dictionary contains the metadata for the time-series plus the gradient (with key “gradient”) and intercept ( with key “intercept”). The gradient and intercept are stored as `pint.Quantity`.

**Return type** Any]

`scmdata.ops.linear_regression_gradient(self, unit=None)`

Calculate gradients of a linear regression of each timeseries

**Parameters** `unit` (`str`) – Output unit for gradients. If not supplied, the gradients’ units will not be converted to a common unit.

**Returns** `self.meta` plus a column with the value of the gradient for each timeseries. The “unit” column is updated to show the unit of the gradient.

**Return type** `pandas.DataFrame`

`scmdata.ops.linear_regression_intercept(self, unit=None)`

Calculate intercepts of a linear regression of each timeseries

---

**Note:** Times in seconds since 1970-01-01 are used as the x-axis for the regressions. Such values can be accessed with `self.time_points.values.astype("datetime64[s]").astype("int")`. This decision does not matter for the gradients, but is important for the intercept values.

---

**Parameters** `unit` (*str*) – Output unit for gradients. If not supplied, the gradients' units will not be converted to a common unit.

**Returns** `self.meta` plus a column with the value of the gradient for each timeseries. The "unit" column is updated to show the unit of the gradient.

**Return type** `pandas.DataFrame`

`scmdata.ops.linear_regression_scmrun(self)`

Re-calculate the timeseries based on a linear regression

**Returns** The timeseries, re-calculated based on a linear regression

**Return type** `scmdata.ScmRun`

`scmdata.ops.multiply(self, other, op_cols, **kwargs)`

Multiply values

**Parameters**

- **other** (*ScmRun*) – *ScmRun* containing data to multiply
- **op\_cols** (*dict of str: str*) – Dictionary containing the columns to drop before multiplying as the keys and the value those columns should hold in the output as the values. For example, if we have `op_cols={"variable": "Emissions|CO2 - Emissions|CO2|Fossil"}` then the multiplication will be performed with an index that uses all columns except the "variable" column and the output will have a "variable" column with the value "Emissions|CO2 - Emissions|CO2|Fossil".
- **\*\*kwargs** (*any*) – Passed to `prep_for_op()`

**Returns** Product of `self` and `other`, using `op_cols` to define the columns which should be dropped before the data is aligned and to define the value of these columns in the output.

**Return type** `ScmRun`

## Examples

```
>>> import numpy as np
>>> from scmdata import ScmRun
>>>
>>> IDX = [2010, 2020, 2030]
>>>
>>> start = ScmRun(
...     data=np.arange(18).reshape(3, 6),
...     index=IDX,
...     columns={
...         "variable": [
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Cumulative Emissions|CO2",
...             "Surface Air Temperature Change",
...         ],
...         "unit": ["GtC / yr", "GtC / yr", "GtC / yr", "GtC / yr", "GtC", "K"],
...         "region": ["World|NH", "World|NH", "World|SH", "World|SH", "World",
...             ↪ "World"],
...     },
... )
```

(continues on next page)



(continued from previous page)

```

...     "model": "idealised",
...     "scenario": "idealised",
... },
... )
>>>
>>> start.head()
time                                2010-01-01 00:00:00
→ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable            unit    region  model    scenario
Emissions|CO2|Fossil  GtC / yr World|NH idealised idealised      0.0
→                6.0                12.0
Emissions|CO2|AFOLU   GtC / yr World|NH idealised idealised      1.0
→                7.0                13.0
Emissions|CO2|Fossil  GtC / yr World|SH idealised idealised      2.0
→                8.0                14.0
Emissions|CO2|AFOLU   GtC / yr World|SH idealised idealised      3.0
→                9.0                15.0
Cumulative Emissions|CO2 GtC      World  idealised idealised      4.0
→                10.0               16.0
>>> fos = start.filter(variable="*Fossil")
>>> fos.head()
time                                2010-01-01 00:00:00
→ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable            unit    region  model    scenario
Emissions|CO2|Fossil GtC / yr World|NH idealised idealised      0.0
→                6.0                12.0
→                World|SH idealised idealised      2.0
→                8.0                14.0
>>>
>>> afolu = start.filter(variable="*AFOLU")
>>> afolu.head()
time                                2010-01-01 00:00:00
→ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable            unit    region  model    scenario
Emissions|CO2|AFOLU GtC / yr World|NH idealised idealised      1.0
→                7.0                13.0
→                World|SH idealised idealised      3.0
→                9.0                15.0
>>>
>>> fos_times_afolu = fos.multiply(
...     afolu, op_cols={"variable": "Emissions|CO2|Fossil : AFOLU"}
... )
>>> fos_times_afolu.head()
time                                2010-01-01 00:00:00  2020-01-01 00:00:00  2030-01-01 00:00:00
model    scenario region  variable            unit
idealised idealised World|NH Emissions|CO2|Fossil : AFOLU gigatC ** 2 / a ** 2
→                0.0                42.0                156.0
→                World|SH Emissions|CO2|Fossil : AFOLU gigatC ** 2 / a ** 2
→                6.0                72.0                210.0

```

scmdata.ops.**prep\_for\_op**(inp, op\_cols, meta, ur=<openscm\_units.\_unit\_registry.ScmUnitRegistry object>)  
 Prepare dataframe for operation

**Parameters**

- **inp** (*ScmRun*) – *ScmRun* containing data to prepare
- **op\_cols** (*dict of str: str*) – Dictionary containing the columns to drop in order to prepare for the operation as the keys (the values are not used). For example, if we have `op_cols={"variable": "Emissions|CO2 - Emissions|CO2|Fossil"}` then we will drop the “variable” column from the index.
- **ur** (*pint.UnitRegistry*) – Pint unit registry to use for the operation

**Returns**

Timeseries to use for the operation. They are the transpose of the normal *ScmRun.timeseries()* output with the columns being Pint arrays (unless “unit” is in `op_cols` in which case no units are available to be used so the columns are standard numpy arrays). We do this so that we can use [Pint’s Pandas interface](#) to handle unit conversions automatically.

**Return type** `pandas.DataFrame`

`scmdata.ops.set_op_values(output, op_cols)`  
Set operation values in output

**Parameters**

- **output** (*pandas.DataFrame*) – Dataframe of which to update the values
- **op\_cols** (*dict of str: str*) – Dictionary containing the columns to update as the keys and the value those columns should hold in the output as the values. For example, if we have `op_cols={"variable": "Emissions|CO2 - Emissions|CO2|Fossil"}` then the output will have a “variable” column with the value “Emissions|CO2 - Emissions|CO2|Fossil”.

**Returns** output with the relevant columns being set according to `op_cols`.

**Return type** `pandas.DataFrame`

`scmdata.ops.subtract(self, other, op_cols, **kwargs)`  
Subtract values

**Parameters**

- **other** (*ScmRun*) – *ScmRun* containing data to subtract
- **op\_cols** (*dict of str: str*) – Dictionary containing the columns to drop before subtracting as the keys and the value those columns should hold in the output as the values. For example, if we have `op_cols={"variable": "Emissions|CO2 - Emissions|CO2|Fossil"}` then the subtraction will be performed with an index that uses all columns except the “variable” column and the output will have a “variable” column with the value “Emissions|CO2 - Emissions|CO2|Fossil”.
- **\*\*kwargs** (*any*) – Passed to `prep_for_op()`

**Returns** Difference between `self` and `other`, using `op_cols` to define the columns which should be dropped before the data is aligned and to define the value of these columns in the output.

**Return type** *ScmRun*

## Examples

```

>>> import numpy as np
>>> from scmdata import ScmRun
>>>
>>> IDX = [2010, 2020, 2030]
>>>
>>> start = ScmRun(
...     data=np.arange(18).reshape(3, 6),
...     index=IDX,
...     columns={
...         "variable": [
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Cumulative Emissions|CO2",
...             "Surface Air Temperature Change",
...         ],
...         "unit": ["GtC / yr", "GtC / yr", "GtC / yr", "GtC / yr", "GtC", "K"],
...         "region": ["World|NH", "World|NH", "World|SH", "World|SH", "World",
...             ↪ "World"],
...         "model": "idealised",
...         "scenario": "idealised",
...     },
... )
>>>
>>> start.head()
time                                     2010-01-01 00:00:00 ↪
↪ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|Fossil  GtC / yr World|NH idealised idealised      0.0 ↪
↪          6.0          12.0
Emissions|CO2|AFOLU   GtC / yr World|NH idealised idealised      1.0 ↪
↪          7.0          13.0
Emissions|CO2|Fossil  GtC / yr World|SH idealised idealised      2.0 ↪
↪          8.0          14.0
Emissions|CO2|AFOLU   GtC / yr World|SH idealised idealised      3.0 ↪
↪          9.0          15.0
Cumulative Emissions|CO2 GtC      World      idealised idealised      4.0 ↪
↪          10.0          16.0
>>> fos = start.filter(variable="*Fossil")
>>> fos.head()
time                                     2010-01-01 00:00:00 ↪
↪ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|Fossil GtC / yr World|NH idealised idealised      0.0 ↪
↪          6.0          12.0
↪          World|SH idealised idealised      2.0 ↪
↪          8.0          14.0
>>>
>>> afolu = start.filter(variable="*AFOLU")

```

(continues on next page)

(continued from previous page)

```

>>> afolu.head()
time                                2010-01-01 00:00:00
→ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model  scenario
Emissions|CO2|AFOLU GtC / yr World|NH idealised idealised      1.0
→              7.0              13.0
→              9.0              15.0
World|SH idealised idealised      3.0
→              15.0

>>>
>>> fos_minus_afolu = fos.subtract(
...     afolu, op_cols={"variable": "Emissions|CO2|Fossil - AFOLU"}
... )
>>> fos_minus_afolu.head()
time                                2010-01-01
→ 00:00:00  2020-01-01 00:00:00  2030-01-01 00:00:00
model      scenario region  variable      unit
idealised idealised World|NH Emissions|CO2|Fossil - AFOLU gigatC / a
→ -1.0              -1.0              -1.0
→              World|SH Emissions|CO2|Fossil - AFOLU gigatC / a
→ -1.0              -1.0              -1.0

>>>
>>> nh_minus_sh = nh.subtract(sh, op_cols={"region": "World|NH - SH"})
>>> nh_minus_sh.head()
time                                2010-01-01
→ 00:00:00  2020-01-01 00:00:00  2030-01-01 00:00:00
model      scenario region  variable      unit
idealised idealised World|NH - SH Emissions|CO2|Fossil gigatC / a
→ 2.0              -2.0              -2.0
→              Emissions|CO2|AFOLU gigatC / a
→ 2.0              -2.0              -2.0

```

### 1.1.11 scmdata.plotting

Plotting helpers for [ScmRun](#)

See the example notebook ‘plotting-with-seaborn.ipynb’ for usage examples

`scmdata.plotting.inject_plotting_methods(cls)`

Inject the plotting methods

**Parameters** `cls` – Target class

`scmdata.plotting.lineplot(self, time_axis=None, **kwargs)`

Make a line plot via [seaborn's lineplot](#)

If only a single unit is present, it will be used as the y-axis label. The axis object is returned so this can be changed by the user if desired.

**Parameters**

- **time\_axis** (`{None, "year", "year-month", "days since 1970-01-01", "seconds since 1970-01-01"}` # noqa: E501) – Time axis to use for the plot.

If `None`, `datetime.datetime` objects will be used.

If `"year"`, the year of each time point will be used.

If "year-month", the year plus (month - 0.5) / 12 will be used.

If "days since 1970-01-01", the number of days since 1st Jan 1970 will be used (calculated using the `datetime` module).

If "seconds since 1970-01-01", the number of seconds since 1st Jan 1970 will be used (calculated using the `datetime` module).

- **\*\*kwargs** – Keyword arguments to be passed to `seaborn.lineplot`. If none are passed, sensible defaults will be used.

**Returns** Output of call to `seaborn.lineplot`

**Return type** `matplotlib.axes._subplots.AxesSubplot`

```
scmdata.plotting.plumeplot(self, ax=None, quantiles_plumes=[((0.05, 0.95), 0.5), ((0.5, 1.0)],
                    hue_var='scenario', hue_label='Scenario', palette=None, style_var='variable',
                    style_label='Variable', dashes=None, linewidth=2, time_axis=None,
                    pre_calculated=False, quantile_over=('ensemble_member',))
```

Make a plume plot, showing plumes for custom quantiles

#### Parameters

- **ax** (`matplotlib.axes._subplots.AxesSubplot`) – Axes on which to make the plot
- **quantiles\_plumes** (`list[tuple[tuple, float]]`) – Configuration to use when plotting quantiles. Each element is a tuple, the first element of which is itself a tuple and the second element of which is the alpha to use for the quantile. If the first element has length two, these two elements are the quantiles to plot and a plume will be made between these two quantiles. If the first element has length one, then a line will be plotted to represent this quantile.
- **hue\_var** (`str`) – The column of `self.meta` which should be used to distinguish different hues.
- **hue\_label** (`str`) – Label to use in the legend for `hue_var`.
- **palette** (`dict`) – Dictionary defining the colour to use for different values of `hue_var`.
- **style\_var** (`str`) – The column of `self.meta` which should be used to distinguish different styles.
- **style\_label** (`str`) – Label to use in the legend for `style_var`.
- **dashes** (`dict`) – Dictionary defining the style to use for different values of `style_var`.
- **linewidth** (`float`) – Width of lines to use (for quantiles which are not to be shown as plumes)
- **time\_axis** (`str`) – Time axis to use for the plot (see `timeseries()`)
- **pre\_calculated** (`bool`) – Are the quantiles pre-calculated? If no, the quantiles will be calculated within this function. Pre-calculating the quantiles using `ScmRun.quantiles_over()` can lead to faster plotting if multiple plots are to be made with the same quantiles.
- **quantile\_over** (`str, tuple[str]`) – Columns of `self.meta` over which the quantiles should be calculated. Only used if `pre_calculated` is `False`.

**Returns** Axes on which the plot was made and the legend items we have made (in case the user wants to move the legend to a different position for example)

**Return type** `matplotlib.axes._subplots.AxesSubplot, list`

## Examples

```
>>> scmrun = ScmRun(  
...     data=np.random.random((10, 3)).T,  
...     columns={  
...         "model": ["a_iam"],  
...         "climate_model": ["a_model"] * 5 + ["a_model_2"] * 5,  
...         "scenario": ["a_scenario"] * 5 + ["a_scenario_2"] * 5,  
...         "ensemble_member": list(range(5)) + list(range(5)),  
...         "region": ["World"],  
...         "variable": ["Surface Air Temperature Change"],  
...         "unit": ["K"],  
...     },  
...     index=[2005, 2010, 2015],  
... )
```

Plot the plumes, calculated over the different ensemble members.

```
>>> scmrun.plumeplot(quantile_over="ensemble_member")
```

Pre-calculate the quantiles, then plot

```
>>> summary_stats = ScmRun(  
...     scmrun.quantiles_over("ensemble_member", quantiles=quantiles)  
... )  
>>> summary_stats.plumeplot(pre_calculated=True)
```

---

**Note:** scmdata is not a plotting library so this function is provided as is, with little testing. In some ways, it is more intended as inspiration for other users than as a robust plotting tool.

---

### 1.1.12 scmdata.processing

Miscellaneous functions for processing scmdata.ScmRun

These functions are intended to be able to be used directly with scmdata.ScmRun.process\_over().

scmdata.processing.calculate\_crossing\_times(*scmrun*, *threshold*, *return\_year=True*)

Calculate the time at which each timeseries crosses a given threshold

#### Parameters

- **scmrun** (scmdata.ScmRun) – Data to calculate the crossing time of
- **threshold** (*float*) – Value to use as the threshold for crossing
- **return\_year** (*bool*) – If True, return the year instead of the datetime

**Returns** Crossing time for scmrun, using the meta of scmrun as the output's index. If the threshold is not crossed, pd.NA is returned.

**Return type** pd.Series

## Notes

This function only returns times that are in the columns of `scmrun`. If you want a finer resolution then you should interpolate your data first. For example, if you have data on a ten-year timestep but want crossing times on an annual resolution, interpolate (or resample) to annual data before calling `calculate_crossing_times`.

`scmdata.processing.calculate_exceedance_probabilities(scmrun, threshold, process_over_cols, output_name=None)`

Calculate exceedance probability over all time

### Parameters

- **scmrun** (`scmdata.ScmRun`) – Ensemble of which to calculate the exceedance probability
- **threshold** (*float*) – Value to use as the threshold for exceedance
- **process\_over\_cols** (*list[str]*) – Columns to not use when grouping the timeseries (typically “run\_id” or “ensemble\_member” or similar)
- **output\_name** (*str*) – If supplied, the name of the output series. If not supplied, “{threshold} exceedance probability” will be used.

**Returns** Exceedance probability over all time over all members of each group in `scmrun`

**Return type** `pd.Series`

**Raises** `ValueError` – `scmrun` has more than one variable or more than one unit (convert to a single unit before calling this function if needed)

## Notes

See the notes of `scmdata.processing.calculate_exceedance_probabilities_over_time()` for an explanation of how the two calculations differ. For most purposes, this is the correct function to use.

`scmdata.processing.calculate_exceedance_probabilities_over_time(scmrun, threshold, process_over_cols, output_name=None)`

Calculate exceedance probability at each point in time

### Parameters

- **scmrun** (`scmdata.ScmRun`) – Ensemble of which to calculate the exceedance probability over time
- **threshold** (*float*) – Value to use as the threshold for exceedance
- **process\_over\_cols** (*list[str]*) – Columns to not use when grouping the timeseries (typically “run\_id” or “ensemble\_member” or similar)
- **output\_name** (*str*) – If supplied, the value to put in the “variable” columns of the output `pd.DataFrame`. If not supplied, “{threshold} exceedance probability” will be used.

**Returns** Timeseries of exceedance probability over time

**Return type** `pd.DataFrame`

**Raises** `ValueError` – `scmrun` has more than one variable or more than one unit (convert to a single unit before calling this function if needed)

## Notes

This differs from `scmdata.processing.calculate_exceedance_probabilities()` because it calculates the exceedance probability at each point in time. That is different from calculating the exceedance probability by first determining the number of ensemble members which cross the threshold **at any point in time** and then dividing by the number of ensemble members. In general, this function will produce a maximum exceedance probability which is equal to or less than the output of `scmdata.processing.calculate_exceedance_probabilities()`. In our opinion, `scmdata.processing.calculate_exceedance_probabilities()` is the correct function to use if you want to know the exceedance probability of a scenario. This function gives a sense of how the exceedance probability evolves over time but, as we said, will generally slightly underestimate the exceedance probability over all time.

`scmdata.processing.calculate_peak(scmrun, output_name=None)`

Calculate peak i.e. maximum of each timeseries

### Parameters

- **scmrun** (`scmdata.ScmRun`) – Ensemble of which to calculate the exceedance probability over time
- **output\_name** (*str*) – If supplied, the value to put in the “variable” columns of the output series. If not supplied, “Peak {variable}” will be used.

**Returns** Peak of each timeseries

**Return type** `pd.Series`

`scmdata.processing.calculate_peak_time(scmrun, output_name=None, return_year=True)`

Calculate peak time i.e. the time at which each timeseries reaches its maximum

### Parameters

- **scmrun** (`scmdata.ScmRun`) – Ensemble of which to calculate the exceedance probability over time
- **output\_name** (*str*) – If supplied, the value to put in the “variable” columns of the output series. If not supplied, “Peak {variable}” will be used.
- **return\_year** (*bool*) – If True, return the year instead of the datetime

**Returns** Peak of each timeseries

**Return type** `pd.Series`

`scmdata.processing.calculate_summary_stats(scmrun, index, exceedance_probabilities_thresholds=(1.5, 2.0, 2.5), exceedance_probabilities_variable='Surface Air Temperature Change', exceedance_probabilities_naming_base=None, peak_quantiles=(0.05, 0.17, 0.5, 0.83, 0.95), peak_variable='Surface Air Temperature Change', peak_naming_base=None, peak_time_naming_base=None, peak_return_year=True, categorisation_variable='Surface Air Temperature Change', categorisation_quantile_cols=('ensemble_member',), progress=False)`

Calculate common summary statistics

### Parameters

- **scmrun** (`scmdata.ScmRun`) – Data of which to calculate the stats



- **index** (*list[str]*) – Columns to use in the index of the output (unit is added if not included)
- **exceedance\_probabilities\_threshold** (*list[float]*) – Thresholds to use for exceedance probabilities
- **exceedance\_probabilities\_variable** (*str*) – Variable to use for exceedance probability calculations
- **exceedance\_probabilities\_naming\_base** (*str*) – String to use as the base for naming the exceedance probabilities. Each exceedance probability output column will have a name given by `exceedance_probabilities_naming_base.format(threshold)` where `threshold` is the exceedance probability threshold to use. If not supplied, the default output of `scmdata.processing.calculate_exceedance_probabilities()` will be used.
- **peak\_quantiles** (*list[float]*) – Quantiles to report in peak calculations
- **peak\_variable** (*str*) – Variable of which to calculate the peak
- **peak\_naming\_base** (*str*) – Base to use for naming the peak outputs. This is combined with the quantile. If not supplied, "{ } peak" is used so the outputs will be named e.g. "0.05 peak", "0.5 peak", "0.95 peak".
- **peak\_time\_naming\_base** (*str*) – Base to use for naming the peak time outputs. This is combined with the quantile. If not supplied, "{ } peak year" is used (unless `peak_return_year` is `False` in which case "{ } peak time" is used) so the outputs will be named e.g. "0.05 peak year", "0.5 peak year", "0.95 peak year".
- **peak\_return\_year** (*bool*) – If `True`, return the year of the peak of `peak_variable`, otherwise return full dates
- **categorisation\_variable** (*str*) – Variable to use for categorisation. Note that this variable point to timeseries that contain global-mean surface air temperatures (GSAT) relative to 1850-1900 (using another reference period will not break this function, but is inconsistent with the original algorithm).
- **categorisation\_quantile\_cols** (*list[str]*) – Columns which represent individual ensemble members in the output (e.g. ["ensemble\_member"]). The quantiles are taking over these columns before the data is passed to `scmdata.processing.categorisation_sr15()`.
- **progress** (*bool*) – Should a progress bar be shown whilst the calculations are done?

**Returns** Summary statistics, with each column being a statistic and the index being given by `index`

**Return type** `pd.DataFrame`

`scmdata.processing.categorisation_sr15(scmrun, index)`

Categorise using the algorithm employed in SR1.5

For more information, see the SR1.5 scenario analysis [notebook](#).

#### Parameters

- **scmrun** – Data to use for the classification. This should contain global-mean surface air temperatures (GSAT) relative to 1850-1900 (using another reference period will not break this function, but is inconsistent with the original algorithm). The data must have a "quantile" column and it must have the 0.33, 0.5 and 0.66 quantiles calculated. This can be done with `scmdata.ScmRun.quantiles_over()`.
- **index** (*list[str]*) – Columns in `scmrun.meta` to use as the index of the output

**Returns** Categorisation of the timeseries

**Return type** class: *pd.Series*

**Raises**

- **ValueError** – More than one variable or one unit is in `scmrun`
- **DimensionalityError** – The units cannot be converted to kelvin

### 1.1.13 `scmdata.run`

```
class scmdata.run.ScRun(data: Any, index: Optional[Any] = None, columns: Optional[Union[Dict[str, list],
Dict[str, str]]] = None, metadata: Optional[Dict[str, Union[str, int, float]]] =
None, copy_data: bool = False, **kwargs: Any)
```

Bases: `scmdata.run.BaseScRun`

Data container for holding one or many time-series of SCM data.

```
__init__(data: Any, index: Optional[Any] = None, columns: Optional[Union[Dict[str, list], Dict[str, str]]]
= None, metadata: Optional[Dict[str, Union[str, int, float]]] = None, copy_data: bool = False,
**kwargs: Any)
```

Initialize the container with timeseries data.

#### Parameters

- **data** (`Union[ScRun, IamDataFrame, pd.DataFrame, np.ndarray, str]`) – If a `ScRun` object is provided, then a new `ScRun` is created with a copy of the values and metadata from `:obj: data`.

A `pandas.DataFrame` with IAMC-format data columns (the result from `ScRun.timeseries()`) can be provided without any additional columns and index information.

If a numpy array of timeseries data is provided, `columns` and `index` must also be specified. The shape of the numpy array should be `(n_times, n_series)` where `n_times` is the number of timesteps and `n_series` is the number of time series.

If a string is passed, data will be attempted to be read from file. Currently, reading from CSV, gzipped CSV and Excel formatted files is supported.

- **index** (`np.ndarray`) – If `index` is not `None`, then the `index` is used as the timesteps for run. All timeseries in the run use the same set of timesteps.

The values will be attempted to be converted to `numpy.datetime[s]` values. Possible input formats include :

- `datetime.datetime`
- `int` Start of year
- `float` Decimal year
- `str` Uses `dateutil.parser()`. Slow and should be avoided if possible

If `index` is `None`, than the time index will be obtained from the data if possible.

- **columns** – If `None`, `ScRun` 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(
    ScmRun(d, columns=col_1).meta,
    ScmRun(d, columns=col_2).meta
)

```

- **metadata** – Optional dictionary of metadata for instance as a whole.

This can be used to store information such as the longer-form information about a particular dataset, for example, dataset description or DOIs.

Defaults to an empty `dict` if no default metadata are provided.

- **copy\_data** (*bool*) – If True, an explicit copy of data is performed.

---

**Note:** The copy can be very expensive on large timeseries and should only be needed in cases where the original data is manipulated.

---

- **\*\*kwargs** – Additional parameters passed to `_read_file()` to read files

#### Raises

- **ValueError** –
  - If you try to load from multiple files at once. If you wish to do this, please use `scmdata.run.run_append()` instead. \* Not specifying index and columns if data is a `numpy.ndarray`
- **scmdata.errors.MissingRequiredColumn** – If metadata for `required_cols` is not found
- **TypeError** – Timeseries cannot be read from data

`add(other, op_cols, **kwargs)`

Add values

#### Parameters

- **other** (*ScmRun*) – *ScmRun* containing data to add
- **op\_cols** (*dict of str: str*) – Dictionary containing the columns to drop before adding as the keys and the value those columns should hold in the output as the values. For example, if we have `op_cols={"variable": "Emissions|CO2 - Emissions|CO2|Fossil"}` then the addition will be performed with an index that uses

all columns except the “variable” column and the output will have a “variable” column with the value “Emissions|CO2 - Emissions|CO2|Fossil”.

- **\*\*kwargs** (any) – Passed to `prep_for_op()`

**Returns** Sum of `self` and `other`, using `op_cols` to define the columns which should be dropped before the data is aligned and to define the value of these columns in the output.

**Return type** *ScmRun*

## Examples

```
>>> import numpy as np
>>> from scmdata import ScmRun
>>>
>>> IDX = [2010, 2020, 2030]
>>>
>>> start = ScmRun(
...     data=np.arange(18).reshape(3, 6),
...     index=IDX,
...     columns={
...         "variable": [
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Cumulative Emissions|CO2",
...             "Surface Air Temperature Change",
...         ],
...         "unit": ["GtC / yr", "GtC / yr", "GtC / yr", "GtC / yr", "GtC", "K"],
...         "region": ["World|NH", "World|NH", "World|SH", "World|SH", "World",
... ↪ "World"],
...         "model": "idealised",
...         "scenario": "idealised",
...     },
... )
>>>
>>> start.head()
time                                     2010-01-01
↪00:00:00  2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit    region  model  scenario
Emissions|CO2|Fossil  GtC / yr World|NH idealised idealised
↪0.0          6.0      12.0
Emissions|CO2|AFOLU   GtC / yr World|NH idealised idealised
↪1.0          7.0      13.0
Emissions|CO2|Fossil  GtC / yr World|SH idealised idealised
↪2.0          8.0      14.0
Emissions|CO2|AFOLU   GtC / yr World|SH idealised idealised
↪3.0          9.0      15.0
Cumulative Emissions|CO2  GtC      World  idealised idealised
↪4.0         10.0      16.0
>>> fos = start.filter(variable="*Fossil")
```

(continues on next page)

(continued from previous page)

```

>>> fos.head()
time                                2010-01-01 00:00:00
↪ 2020-01-01 00:00:00 2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|Fossil GtC / yr World|NH idealised idealised      0.0
↪              6.0              12.0
↪              8.0              14.0
>>>
>>> afolu = start.filter(variable="*AFOLU")
>>> afolu.head()
time                                2010-01-01 00:00:00
↪ 2020-01-01 00:00:00 2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|AFOLU GtC / yr World|NH idealised idealised      1.0
↪              7.0              13.0
↪              9.0              15.0
>>>
>>> total = fos.add(afolu, op_cols={"variable": "Emissions|CO2"})
>>> total.head()
time                                2010-01-01 00:00:00
↪ 2020-01-01 00:00:00 2030-01-01 00:00:00
model      scenario region  variable      unit
idealised idealised World|NH Emissions|CO2 gigatC / a      1.0
↪              13.0              25.0
↪              17.0              29.0
>>>
>>> nh = start.filter(region="*NH")
>>> nh.head()
time                                2010-01-01 00:00:00
↪ 2020-01-01 00:00:00 2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|Fossil GtC / yr World|NH idealised idealised      0.0
↪              6.0              12.0
Emissions|CO2|AFOLU  GtC / yr World|NH idealised idealised      1.0
↪              7.0              13.0
>>>
>>> sh = start.filter(region="*SH")
>>> sh.head()
time                                2010-01-01 00:00:00
↪ 2020-01-01 00:00:00 2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|Fossil GtC / yr World|SH idealised idealised      2.0
↪              8.0              14.0
Emissions|CO2|AFOLU  GtC / yr World|SH idealised idealised      3.0
↪              9.0              15.0
>>>
>>> world = nh.add(sh, op_cols={"region": "World"})
>>> world.head()
time                                2010-01-01 00:00:00
↪ 2020-01-01 00:00:00 2030-01-01 00:00:00

```

(continues on next page)

(continued from previous page)

model	scenario	region	variable	unit	
idealised	idealised	World	Emissions CO2 Fossil	gigatC / a	2.0
↔		14.0	26.0		
			Emissions CO2 AFOLU	gigatC / a	4.0
↔		16.0	28.0		

**adjust\_median\_to\_target**(*target*, *evaluation\_period*, *process\_over*=None, *check\_groups\_identical*=False, *check\_groups\_identical\_kwargs*=None)

Adjust the median of (an ensemble of) timeseries to a specified target

#### Parameters

- **target** (*float*) – Value to which the median of each (group of) timeseries should be adjusted
- **evaluation\_period** (*list[int]*) – Period over which the median should be evaluated
- **process\_over** (*list*) – Metadata to treat as ‘ensemble members’ i.e. all other columns in the metadata of *self* will be used to group the timeseries before calculating the median. If not supplied, timeseries will not be grouped.
- **check\_groups\_identical** (*bool*) – Should we check that the median of each group is the same before making the adjustment?
- **check\_groups\_identical\_kwargs** (*dict*) – Only used if *check\_groups\_identical* is True, in which case these are passed through to *np.testing.assert\_allclose*

#### Raises

- **NotImplementedError** – *evaluation\_period* is based on times not years
- **AssertionError** – If *check\_groups\_identical* is True and the median of each group is not the same before making the adjustment.

**Returns** Timeseries adjusted to have the intended median

**Return type** *ScmRun*

**append**(*other*, *inplace*: *bool* = False, *duplicate\_msg*: *Union[str, bool]* = True, *metadata*: *Optional[Dict[str, Union[str, int, float]]]* = None, *\*\*kwargs*: *Any*)

Append additional data to the current data.

For details, see [run\\_append\(\)](#).

#### Parameters

- **other** – Data (in format which can be cast to *ScmRun*) to append
- **inplace** – If True, append data in place and return None. Otherwise, return a new *ScmRun* instance with the appended data.
- **duplicate\_msg** – If True, raise a *scmdata.errors.NonUniqueMetadataError* error so the user can see the duplicate timeseries. If False, take the average and do not raise a warning or error. If "warn", raise a warning if duplicate data is detected.
- **metadata** – If not None, override the metadata of the resulting *ScmRun* with metadata. Otherwise, the metadata for the runs are merged. In the case where there are duplicate metadata keys, the values from the first run are used.
- **\*\*kwargs** – Keywords to pass to *ScmRun.\_\_init\_\_()* when reading *other*

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

**Return type** *ScmRun*

**Raises** *NonUniqueMetadataError* – If the appending results in timeseries with duplicate meta-data and `duplicate_msg` is True

**append\_timewise**(*other*, *align\_columns*)  
Append timeseries along the time axis

**Parameters**

- **other** (*scmdata.ScmRun*) – *scmdata.ScmRun* containing the timeseries to append
- **align\_columns** (*list*) – Columns used to align *other* and *self* when joining

**Returns** Result of joining *self* and *other* along the time axis

**Return type** *scmdata.ScmRun*

**convert\_unit**(*unit*: *str*, *context*: *Optional[str]* = None, *inplace*: *bool* = False, *\*\*kwargs*: *Any*)  
Convert the units of a selection of timeseries.

Uses *scmdata.units.UnitConverter* to perform the conversion.

**Parameters**

- **unit** – Unit to convert to. This must be recognised by *UnitConverter*.
- **context** – Context to use for the conversion i.e. which metric to apply when performing CO<sub>2</sub>-equivalent calculations. If None, no metric will be applied and CO<sub>2</sub>-equivalent calculations will raise *DimensionalityError*.
- **inplace** – If True, apply the conversion inplace and return None
- **\*\*kwargs** – Extra arguments which are passed to *filter()* to limit the timeseries which are attempted to be converted. Defaults to selecting the entire *ScmRun*, which will likely fail.

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

**Return type** *ScmRun*

## Notes

If *context* is not None, then the context used for the conversion will be checked against any existing metadata and, if the conversion is valid, stored in the output's metadata.

**Raises** *ValueError* – "unit\_context" is already included in *self*'s *meta\_attributes()* and it does not match context for the variables to be converted.

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

Also creates copies the underlying Timeseries data

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

**Return type** *ScmRun*

**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 CO<sub>2</sub> for energy from coal would be "Emissions|CO<sub>2</sub>|Energy|Coal".

**Type** *str*

**delta\_per\_delta\_time**(*out\_var=None*)

Calculate change in timeseries values for each timestep, divided by the size of the timestep

The output is placed on the middle of each timestep and is one timestep shorter than the input.

**Parameters** **out\_var** (*str*) – If provided, the variable column of the output is set equal to *out\_var*. Otherwise, the output variables are equal to the input variables, prefixed with “Delta ”.

**Returns** *scmdata.ScmRun* containing the changes in values of *self*, normalised by the change in time

**Return type** *scmdata.ScmRun*

**Warns** **UserWarning** – The data contains nans. If this happens, the output data will also contain nans.

**divide**(*other, op\_cols, \*\*kwargs*)

Divide values (*self* / *other*)

**Parameters**

- **other** (*ScmRun*) – *ScmRun* containing data to divide
- **op\_cols** (*dict of str: str*) – Dictionary containing the columns to drop before dividing as the keys and the value those columns should hold in the output as the values. For example, if we have *op\_cols*={“variable”: “Emissions|CO2 - Emissions|CO2|Fossil”} then the division will be performed with an index that uses all columns except the “variable” column and the output will have a “variable” column with the value “Emissions|CO2 - Emissions|CO2|Fossil”.
- **\*\*kwargs** (*any*) – Passed to *prep\_for\_op()*

**Returns** Quotient of *self* and *other*, using *op\_cols* to define the columns which should be dropped before the data is aligned and to define the value of these columns in the output.

**Return type** *ScmRun*

## Examples

```
>>> import numpy as np
>>> from scmdata import ScmRun
>>>
>>> IDX = [2010, 2020, 2030]
>>>
>>> start = ScmRun(
...     data=np.arange(18).reshape(3, 6),
...     index=IDX,
...     columns={
...         "variable": [
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Cumulative Emissions|CO2",
...             "Surface Air Temperature Change",
...         ],
```

(continues on next page)



(continued from previous page)

```

...     "unit": ["GtC / yr", "GtC / yr", "GtC / yr", "GtC / yr", "GtC", "K
↪"],
...     "region": ["World|NH", "World|NH", "World|SH", "World|SH", "World",
↪"World"],
...     "model": "idealised",
...     "scenario": "idealised",
...     },
... )
>>>
>>> start.head()
time                                     2010-01-01↪
↪00:00:00  2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model    scenario
Emissions|CO2|Fossil  GtC / yr World|NH idealised idealised
↪0.0          6.0          12.0
Emissions|CO2|AFOLU   GtC / yr World|NH idealised idealised
↪1.0          7.0          13.0
Emissions|CO2|Fossil  GtC / yr World|SH idealised idealised
↪2.0          8.0          14.0
Emissions|CO2|AFOLU   GtC / yr World|SH idealised idealised
↪3.0          9.0          15.0
Cumulative Emissions|CO2 GtC      World  idealised idealised
↪4.0          10.0         16.0
>>> fos = start.filter(variable="*Fossil")
>>> fos.head()
time                                     2010-01-01 00:00:00↪
↪ 2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model    scenario
Emissions|CO2|Fossil GtC / yr World|NH idealised idealised      0.0↪
↪          6.0          12.0
          World|SH idealised idealised      2.0↪
↪          8.0          14.0
>>>
>>> afolu = start.filter(variable="*AFOLU")
>>> afolu.head()
time                                     2010-01-01 00:00:00 ↪
↪2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model    scenario
Emissions|CO2|AFOLU GtC / yr World|NH idealised idealised      1.0 ↪
↪          7.0          13.0
          World|SH idealised idealised      3.0 ↪
↪          9.0          15.0
>>>
>>> fos_afolu_ratio = fos.divide(
...     afolu, op_cols={"variable": "Emissions|CO2|Fossil : AFOLU"}
... )
>>> fos_afolu_ratio.head()
time                                     2010-
↪01-01 00:00:00  2020-01-01 00:00:00  2030-01-01 00:00:00
model    scenario region  variable      unit
idealised idealised World|NH Emissions|CO2|Fossil : AFOLU dimensionless
↪      0.000000      0.857143      0.923077

```

(continues on next page)

(continued from previous page)

	World SH Emissions CO2 Fossil : AFOLU dimensionless	
↩ 0.666667	0.888889	0.933333 ↪

**drop\_meta**(columns: Union[list, str], inplace: Optional[bool] = False)

Drop meta columns out of the Run

**Parameters**

- **columns** – The column or columns to drop
- **inplace** – If True, do operation inplace and return None.

**Raises** **KeyError** – If any of the columns do not exist in the meta DataFrame**property empty:** boolIndicate whether *ScmRun* is empty i.e. contains no data**Returns** If *ScmRun* is empty, return True, if not return False**Return type** bool**filter**(keep: bool = True, inplace: bool = False, log\_if\_empty: bool = True, \*\*kwargs: Any)

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

```

>>> df
<scmdata.ScmRun (timeseries: 3, timepoints: 3)>
Time:
  Start: 2005-01-01T00:00:00
  End: 2015-01-01T00:00:00
Meta:
   model  scenario region  variable  unit climate_model
0  a_iam  a_scenario  World  Primary Energy  EJ/yr      a_model
1  a_iam  a_scenario  World  Primary Energy|Coal  EJ/yr      a_model
2  a_iam  a_scenario2  World  Primary Energy  EJ/yr      a_model
[3 rows x 7 columns]

>>> df.filter(scenario="a_scenario")
<scmdata.ScmRun (timeseries: 2, timepoints: 3)>
Time:
  Start: 2005-01-01T00:00:00
  End: 2015-01-01T00:00:00
Meta:
   model  scenario region  variable  unit climate_model
0  a_iam  a_scenario  World  Primary Energy  EJ/yr      a_model
1  a_iam  a_scenario  World  Primary Energy|Coal  EJ/yr      a_model
[2 rows x 7 columns]

>>> df.filter(scenario="a_scenario", keep=False)
<scmdata.ScmRun (timeseries: 1, timepoints: 3)>
Time:
  Start: 2005-01-01T00:00:00
  End: 2015-01-01T00:00:00
Meta:
   model  scenario region  variable  unit climate_model
2  a_iam  a_scenario2  World  Primary Energy  EJ/yr      a_model
[1 rows x 7 columns]

```

(continues on next page)

(continued from previous page)

```
>>> df.filter(level=1)
<scmdata.ScmRun (timeseries: 2, timepoints: 3)>
Time:
  Start: 2005-01-01T00:00:00
  End: 2015-01-01T00:00:00
Meta:
   model    scenario region    variable    unit climate_model
0  a_iam  a_scenario  World  Primary Energy  EJ/yr      a_model
2  a_iam  a_scenario2 World  Primary Energy  EJ/yr      a_model
[2 rows x 7 columns]
```

```
>>> df.filter(year=range(2000, 2011))
<scmdata.ScmRun (timeseries: 3, timepoints: 2)>
Time:
  Start: 2005-01-01T00:00:00
  End: 2010-01-01T00:00:00
Meta:
   model    scenario region    variable    unit climate_model
0  a_iam  a_scenario  World  Primary Energy  EJ/yr      a_model
1  a_iam  a_scenario  World  Primary Energy|Coal  EJ/yr      a_model
2  a_iam  a_scenario2 World  Primary Energy  EJ/yr      a_model
[2 rows x 7 columns]
```

### Parameters

- **keep** – If True, keep all timeseries satisfying the filters, otherwise drop all the timeseries satisfying the filters
- **inplace** – If True, do operation inplace and return None
- **log\_if\_empty** – If True, log a warning level message if the result is empty.
- **\*\*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 `regex=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** `ScmRun`

**classmethod** `from_nc(fname)`

Read a netCDF4 file from disk

**Parameters** `fname` (`str`) – Filename to read

See also:

`scmdata.run.ScmRun.from_nc()`

**get\_meta\_columns\_except**(\*not\_group)

Get columns in meta except a set

**Parameters** **not\_group** (*str or list of str*) – Columns to exclude from the grouping

**Returns** Meta columns except the ones supplied (sorted alphabetically)

**Return type** `list`

**get\_unique\_meta**(*meta: str, no\_duplicates: Optional[bool] = False*) → Union[List[Any], Any]

Get unique values in a metadata column.

**Parameters**

- **meta** – Column to retrieve metadata for
- **no\_duplicates** – Should I raise an error if there is more than one unique value in the metadata column?

**Raises**

- **ValueError** – There is more than one unique value in the metadata column and `no_duplicates` is `True`.
- **KeyError** – If a meta column does not exist in the run's metadata

**Returns** List of unique metadata values. If `no_duplicates` is `True` the metadata value will be returned (rather than a list).

**Return type** [List[Any], Any]

**groupby**(\*group)

Group the object by unique metadata

Enables iteration over groups of data. For example, to iterate over each scenario in the object

```
>>> for group in df.groupby("scenario"):
>>>     print(group)
<scmdata.ScmRun (timeseries: 2, timepoints: 3)>
Time:
  Start: 2005-01-01T00:00:00
  End: 2015-01-01T00:00:00
Meta:
   model  scenario region      variable  unit climate_model
0  a_iam  a_scenario World  Primary Energy  EJ/yr      a_model
1  a_iam  a_scenario World  Primary Energy|Coal  EJ/yr      a_model
<scmdata.ScmRun (timeseries: 1, timepoints: 3)>
Time:
  Start: 2005-01-01T00:00:00
  End: 2015-01-01T00:00:00
Meta:
   model  scenario region      variable  unit climate_model
2  a_iam  a_scenario2 World  Primary Energy  EJ/yr      a_model
```

**Parameters** **group** (*str or list of str*) – Columns to group by

**Returns** See the documentation for `RunGroupBy` for more information

**Return type** `RunGroupBy`

**groupby\_all\_except**(*\*not\_group*)

Group the object by unique metadata apart from the input columns

In other words, the groups are determined by all columns in `self.meta` except for those in `not_group`

**Parameters** `not_group` (*str or list of str*) – Columns to exclude from the grouping

**Returns** See the documentation for `RunGroupBy` for more information

**Return type** `RunGroupBy`

**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** `pandas.DataFrame`

**integrate**(*out\_var=None*)

Integrate with respect to time

**Parameters** `out_var` (*str*) – If provided, the variable column of the output is set equal to `out_var`. Otherwise, the output variables are equal to the input variables, prefixed with “Cumulative ”.

**Returns** `scmdata.ScmRun` containing the integral of `self` with respect to time

**Return type** `scmdata.ScmRun`

**Warns** `UserWarning` – The data being integrated contains nans. If this happens, the output data will also contain nans.

**interpolate**(*target\_times: Union[numpy.ndarray, List[Union[datetime.datetime, int]]], interpolation\_type: str = 'linear', extrapolation\_type: str = 'linear'*)

Interpolate the data onto a new time frame.

**Parameters**

- **target\_times** – Time grid onto which to interpolate
- **interpolation\_type** (*str*) – Interpolation type. Options are ‘linear’
- **extrapolation\_type** (*str or None*) – Extrapolation type. Options are None, ‘linear’ or ‘constant’

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

**Return type** `ScmRun`

**line\_plot**(*\*\*kwargs*)

Make a line plot via `seaborn’s lineplot`

Deprecated: use `lineplot()` instead

**Parameters** **\*\*kwargs** – Keyword arguments to be passed to `seaborn.lineplot`. If none are passed, sensible defaults will be used.

**Returns** Output of call to `seaborn.lineplot`

**Return type** `matplotlib.axes._subplots.AxesSubplot`

**linear\_regression()**

Calculate linear regression of each timeseries

---

**Note:** Times in seconds since 1970-01-01 are used as the x-axis for the regressions. Such values can be accessed with `self.time_points.values.astype("datetime64[s]").astype("int")`. This decision does not matter for the gradients, but is important for the intercept values.

---

**Returns** list of dict[str] – List of dictionaries. Each dictionary contains the metadata for the timeseries plus the gradient (with key "gradient") and intercept (with key "intercept"). The gradient and intercept are stored as `pint.Quantity`.

**Return type** Any]

**linear\_regression\_gradient(unit=None)**

Calculate gradients of a linear regression of each timeseries

**Parameters** unit (str) – Output unit for gradients. If not supplied, the gradients' units will not be converted to a common unit.

**Returns** self.meta plus a column with the value of the gradient for each timeseries. The "unit" column is updated to show the unit of the gradient.

**Return type** pandas.DataFrame

**linear\_regression\_intercept(unit=None)**

Calculate intercepts of a linear regression of each timeseries

---

**Note:** Times in seconds since 1970-01-01 are used as the x-axis for the regressions. Such values can be accessed with `self.time_points.values.astype("datetime64[s]").astype("int")`. This decision does not matter for the gradients, but is important for the intercept values.

---

**Parameters** unit (str) – Output unit for gradients. If not supplied, the gradients' units will not be converted to a common unit.

**Returns** self.meta plus a column with the value of the gradient for each timeseries. The "unit" column is updated to show the unit of the gradient.

**Return type** pandas.DataFrame

**linear\_regression\_scmrun()**

Re-calculate the timeseries based on a linear regression

**Returns** The timeseries, re-calculated based on a linear regression

**Return type** scmdata.ScmRun

**lineplot(time\_axis=None, \*\*kwargs)**

Make a line plot via `seaborn's lineplot`

If only a single unit is present, it will be used as the y-axis label. The axis object is returned so this can be changed by the user if desired.

**Parameters**

- **time\_axis** ({None, "year", "year-month", "days since 1970-01-01", "seconds since 1970-01-01"} # noqa: E501) – Time axis to use for the plot.

If None, `datetime.datetime` objects will be used.

If "year", the year of each time point will be used.

If "year-month", the year plus (month - 0.5) / 12 will be used.

If "days since 1970-01-01", the number of days since 1st Jan 1970 will be used (calculated using the `datetime` module).

If "seconds since 1970-01-01", the number of seconds since 1st Jan 1970 will be used (calculated using the `datetime` module).

- **\*\*kwargs** – Keyword arguments to be passed to `seaborn.lineplot`. If none are passed, sensible defaults will be used.

**Returns** Output of call to `seaborn.lineplot`

**Return type** `matplotlib.axes._subplots.AxesSubplot`

**long\_data**(*time\_axis=None*)

Return data in long form, particularly useful for plotting with seaborn

**Parameters** **time\_axis** (`{None, "year", "year-month", "days since 1970-01-01", "seconds since 1970-01-01"}`) – Time axis to use for the output's columns.

If None, `datetime.datetime` objects will be used.

If "year", the year of each time point will be used.

If "year-month", the year plus (month - 0.5) / 12 will be used.

If "days since 1970-01-01", the number of days since 1st Jan 1970 will be used (calculated using the `datetime` module).

If "seconds since 1970-01-01", the number of seconds since 1st Jan 1970 will be used (calculated using the `datetime` module).

**Returns** `pandas.DataFrame` containing the data in 'long form' (i.e. one observation per row).

**Return type** `pandas.DataFrame`

**property meta:** `pandas.core.frame.DataFrame`

Metadata

**property meta\_attributes**

Get a list of all meta keys

**Returns** Sorted list of meta keys

**Return type** `list`

**multiply**(*other, op\_cols, \*\*kwargs*)

Multiply values

**Parameters**

- **other** (`ScmRun`) – `ScmRun` containing data to multiply
- **op\_cols** (*dict of str: str*) – Dictionary containing the columns to drop before multiplying as the keys and the value those columns should hold in the output as the values. For example, if we have `op_cols={"variable": "Emissions|CO2 - Emissions|CO2|Fossil"}` then the multiplication will be performed with an index that uses all columns except the "variable" column and the output will have a "variable" column with the value "Emissions|CO2 - Emissions|CO2|Fossil".
- **\*\*kwargs** (*any*) – Passed to `prep_for_op()`

**Returns** Product of self and other, using `op_cols` to define the columns which should be dropped before the data is aligned and to define the value of these columns in the output.

**Return type** *ScmRun*

## Examples

```
>>> import numpy as np
>>> from scmdata import ScmRun
>>>
>>> IDX = [2010, 2020, 2030]
>>>
>>> start = ScmRun(
...     data=np.arange(18).reshape(3, 6),
...     index=IDX,
...     columns={
...         "variable": [
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Cumulative Emissions|CO2",
...             "Surface Air Temperature Change",
...         ],
...         "unit": ["GtC / yr", "GtC / yr", "GtC / yr", "GtC / yr", "GtC", "K"],
...         "region": ["World|NH", "World|NH", "World|SH", "World|SH", "World", "World"],
...         "model": "idealised",
...         "scenario": "idealised",
...     },
... )
>>>
>>> start.head()
```

time						2010-01-01
00:00:00	2020-01-01 00:00:00	2030-01-01 00:00:00				
variable	unit	region	model	scenario		
Emissions CO2 Fossil	GtC / yr	World NH	idealised	idealised		
0.0	6.0		12.0			
Emissions CO2 AFOLU	GtC / yr	World NH	idealised	idealised		
1.0	7.0		13.0			
Emissions CO2 Fossil	GtC / yr	World SH	idealised	idealised		
2.0	8.0		14.0			
Emissions CO2 AFOLU	GtC / yr	World SH	idealised	idealised		
3.0	9.0		15.0			
Cumulative Emissions CO2	GtC	World	idealised	idealised		
4.0	10.0		16.0			

```
>>> fos = start.filter(variable="*Fossil")
>>> fos.head()
```

time						2010-01-01 00:00:00
2020-01-01 00:00:00	2030-01-01 00:00:00					
variable	unit	region	model	scenario		

(continues on next page)



(continued from previous page)

```

Emissions|CO2|Fossil GtC / yr World|NH idealised idealised      0.0
↪      6.0      12.0
      World|SH idealised idealised      2.0
↪      8.0      14.0
>>>
>>> afolu = start.filter(variable="*AFOLU")
>>> afolu.head()
time      2010-01-01 00:00:00
↪ 2020-01-01 00:00:00 2030-01-01 00:00:00
variable      unit      region      model      scenario
Emissions|CO2|AFOLU GtC / yr World|NH idealised idealised      1.0
↪      7.0      13.0
      World|SH idealised idealised      3.0
↪      9.0      15.0
>>>
>>> fos_times_afolu = fos.multiply(
...     afolu, op_cols={"variable": "Emissions|CO2|Fossil : AFOLU"}
... )
>>> fos_times_afolu.head()
time      2010-01-01 00:00:00 2020-01-01 00:00:00 2030-01-01 00:00:00
model      scenario      region      variable      unit
idealised idealised World|NH Emissions|CO2|Fossil : AFOLU gigatC ** 2 / a ** 2
↪      0.0      42.0      156.0
      World|SH Emissions|CO2|Fossil : AFOLU gigatC ** 2 / a ** 2
↪      6.0      72.0      210.0

```

```

plumeplot(ax=None, quantiles_plumes=[((0.05, 0.95), 0.5), ((0.5, ), 1.0)], hue_var='scenario',
          hue_label='Scenario', palette=None, style_var='variable', style_label='Variable', dashes=None,
          linewidth=2, time_axis=None, pre_calculated=False, quantile_over=('ensemble_member',))

```

Make a plume plot, showing plumes for custom quantiles

#### Parameters

- **ax** (`matplotlib.axes._subplots.AxesSubplot`) – Axes on which to make the plot
- **quantiles\_plumes** (`list[tuple[tuple, float]]`) – Configuration to use when plotting quantiles. Each element is a tuple, the first element of which is itself a tuple and the second element of which is the alpha to use for the quantile. If the first element has length two, these two elements are the quantiles to plot and a plume will be made between these two quantiles. If the first element has length one, then a line will be plotted to represent this quantile.
- **hue\_var** (`str`) – The column of `self.meta` which should be used to distinguish different hues.
- **hue\_label** (`str`) – Label to use in the legend for `hue_var`.
- **palette** (`dict`) – Dictionary defining the colour to use for different values of `hue_var`.
- **style\_var** (`str`) – The column of `self.meta` which should be used to distinguish different styles.
- **style\_label** (`str`) – Label to use in the legend for `style_var`.
- **dashes** (`dict`) – Dictionary defining the style to use for different values of `style_var`.

- **linewidth** (*float*) – Width of lines to use (for quantiles which are not to be shown as plumes)
- **time\_axis** (*str*) – Time axis to use for the plot (see [timeseries\(\)](#))
- **pre\_calculated** (*bool*) – Are the quantiles pre-calculated? If no, the quantiles will be calculated within this function. Pre-calculating the quantiles using [ScmRun.quantiles\\_over\(\)](#) can lead to faster plotting if multiple plots are to be made with the same quantiles.
- **quantile\_over** (*str*, *tuple[str]*) – Columns of `self.meta` over which the quantiles should be calculated. Only used if `pre_calculated` is `False`.

**Returns** Axes on which the plot was made and the legend items we have made (in case the user wants to move the legend to a different position for example)

**Return type** `matplotlib.axes._subplots.AxesSubplot`, `list`

## Examples

```
>>> scmrn = ScmRun(  
...     data=np.random.random((10, 3)).T,  
...     columns={  
...         "model": ["a_iam"],  
...         "climate_model": ["a_model"] * 5 + ["a_model_2"] * 5,  
...         "scenario": ["a_scenario"] * 5 + ["a_scenario_2"] * 5,  
...         "ensemble_member": list(range(5)) + list(range(5)),  
...         "region": ["World"],  
...         "variable": ["Surface Air Temperature Change"],  
...         "unit": ["K"],  
...     },  
...     index=[2005, 2010, 2015],  
... )
```

Plot the plumes, calculated over the different ensemble members.

```
>>> scmrn.plumeplot(quantile_over="ensemble_member")
```

Pre-calculate the quantiles, then plot

```
>>> summary_stats = ScmRun(  
...     scmrn.quantiles_over("ensemble_member", quantiles=quantiles)  
... )  
>>> summary_stats.plumeplot(pre_calculated=True)
```

---

**Note:** `scmdata` is not a plotting library so this function is provided as is, with little testing. In some ways, it is more intended as inspiration for other users than as a robust plotting tool.

---

**process\_over** (*cols*: `Union[str, List[str]]`, *operation*: `Union[str, Callable[[pandas.core.frame.DataFrame], Union[pandas.core.frame.DataFrame, pandas.core.series.Series, float]]]`, *na\_override*=`1000000.0`, *\*\*kwargs*: `Any`) → `pandas.core.frame.DataFrame`

Process the data over the input columns.

### Parameters

- **cols** – Columns to perform the operation on. The timeseries will be grouped by all other columns in *meta*.
- **operation** (*str* or *func*) – The operation to perform.

If a string is provided, the equivalent pandas groupby function is used. Note that not all groupby functions are available as some do not make sense for this particular application. Additional information about the arguments for the pandas groupby functions can be found at <https://pandas.pydata.org/pandas-docs/stable/reference/groupby.html>.

If a function is provided, it will be applied to each group. The function must take a dataframe as its first argument and return a DataFrame, Series or scalar.

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 the same as taking the median and not the same as taking the mean/average.

- **na\_override** (*[int, float]*) – Convert any nan value in the timeseries meta to this value during processing. The meta values converted back to nan's before the run is returned. This should not need to be changed unless the existing metadata clashes with the default `na_override` value.

This functionality is disabled if `na_override` is `None`, but may result in incorrect results if the timeseries meta includes any nan's.

- **\*\*kwargs** – Keyword arguments to pass `operation` (or the pandas operation if `operation` is a string)

**Returns** The result of `operation`, grouped by all columns in *meta* other than `cols`

**Return type** `pandas.DataFrame`

**Raises** `ValueError` – If the operation is not an allowed operation

If the value of `na_override` clashes with any existing metadata

**quantiles\_over**(*cols: Union[str, List[str]]*, *quantiles: Union[str, List[float]]*, *\*\*kwargs: Any*) → `pandas.core.frame.DataFrame`

Calculate quantiles of the data over the input columns.

#### Parameters

- **cols** – Columns to perform the operation on. The timeseries will be grouped by all other columns in *meta*.
- **quantiles** – The quantiles to calculate. This should be a list of quantiles to calculate (quantile values between 0 and 1). `quantiles` can also include the strings “median” or “mean” if these values are to be calculated.
- **\*\*kwargs** – Passed to `process_over()`.

**Returns** The quantiles of the timeseries, grouped by all columns in *meta* other than `cols`. Each calculated quantile is given a label which is stored in the `quantile` column within the output index.

**Return type** `pandas.DataFrame`

**Raises** `TypeError` – `operation` is included in `kwargs`. The operation is inferred from `quantiles`.

**reduce**(*func*, *dim=None*, *axis=None*, *\*\*kwargs*)

Apply a function along a given axis

This is to provide the GroupBy functionality in `ScmRun.groupby()` and is not generally called directly.

This implementation is very bare-bones - no reduction along the time dimension is allowed and only the `dim` parameter is used.

#### Parameters

- **func** (*function*) –
- **dim** (*str*) – Ignored
- **axis** (*int*) – The dimension along which the function is applied. The only valid value is 0 which corresponds to the along the time-series dimension.
- **kwargs** – Other parameters passed to *func*

#### Returns

Return type `ScmRun`

#### Raises

- **ValueError** – If a dimension other than None is provided
- **NotImplementedError** – If *axis* is anything other than 0

**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** – Deprecated
- **\*\*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** New object containing the timeseries, adjusted to the reference period mean. The reference period year bounds are stored in the meta columns "reference\_period\_start\_year" and "reference\_period\_end\_year".

Return type `ScmRun`

Raises **NotImplementedError** – `append_str` is not None

**required\_cols** = ('model', 'scenario', 'region', 'variable', 'unit')

Minimum metadata columns required by an `ScmRun`.

If an application requires a different set of required metadata, this can be specified by overriding `required_cols` on a custom class inheriting `scmdata.run.BaseScmRun`. Note that at a minimum, ("variable", "unit") columns are required.

**resample**(*rule: str = 'AS', \*\*kwargs: Any*)

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** – 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 `ScmRun` instance on a new time index

Return type *ScmRun*

## Examples

Resample a run to annual values

```
>>> scm_df = ScmRun(
...     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
year
2000          1
2010         10
```

An annual timeseries can be 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   10.000000
```

```
>>> m_df = scm_df.resample('MS')
>>> m_df.timeseries().T
model          a_iam
scenario      a_scenario
region        World
variable      Primary Energy
```

(continues on next page)

(continued from previous page)

unit	EJ/y
time	
2000-01-01 00:00:00	1.0000000
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	10.0000000
[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.

**property shape:** `tuple`

Get the shape of the underlying data as (num\_timeseries, num\_timesteps)

### Returns

**Return type** tuple of int

**subtract**(*other*, *op\_cols*, *\*\*kwargs*)

Subtract values

### Parameters

- **other** (*ScmRun*) – *ScmRun* containing data to subtract
- **op\_cols** (*dict of str: str*) – Dictionary containing the columns to drop before subtracting as the keys and the value those columns should hold in the output as the values. For example, if we have `op_cols={"variable": "Emissions|CO2 - Emissions|CO2|Fossil"}` then the subtraction will be performed with an index that uses all columns except the “variable” column and the output will have a “variable” column with the value “Emissions|CO2 - Emissions|CO2|Fossil”.
- **\*\*kwargs** (*any*) – Passed to `prep_for_op()`

**Returns** Difference between `self` and `other`, using `op_cols` to define the columns which should be dropped before the data is aligned and to define the value of these columns in the output.

Return type *ScmRun*

## Examples

```
>>> import numpy as np
>>> from scmdata import ScmRun
>>>
>>> IDX = [2010, 2020, 2030]
>>>
>>> start = ScmRun(
...     data=np.arange(18).reshape(3, 6),
...     index=IDX,
...     columns={
...         "variable": [
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Emissions|CO2|Fossil",
...             "Emissions|CO2|AFOLU",
...             "Cumulative Emissions|CO2",
...             "Surface Air Temperature Change",
...         ],
...         "unit": ["GtC / yr", "GtC / yr", "GtC / yr", "GtC / yr", "GtC", "K
→"],
...         "region": ["World|NH", "World|NH", "World|SH", "World|SH", "World",
→"World"],
...         "model": "idealised",
...         "scenario": "idealised",
...     },
... )
>>>
>>> start.head()
```

time					2010-01-01
→ 00:00:00	2020-01-01 00:00:00	2030-01-01 00:00:00			
variable	unit	region	model	scenario	
Emissions CO2 Fossil	GtC / yr	World NH	idealised	idealised	
→ 0.0	6.0		12.0		
Emissions CO2 AFOLU	GtC / yr	World NH	idealised	idealised	
→ 1.0	7.0		13.0		
Emissions CO2 Fossil	GtC / yr	World SH	idealised	idealised	
→ 2.0	8.0		14.0		
Emissions CO2 AFOLU	GtC / yr	World SH	idealised	idealised	
→ 3.0	9.0		15.0		
Cumulative Emissions CO2	GtC	World	idealised	idealised	
→ 4.0	10.0		16.0		

```
>>> fos = start.filter(variable="*Fossil")
>>> fos.head()
```

time					2010-01-01 00:00:00
→ 2020-01-01 00:00:00	2030-01-01 00:00:00				
variable	unit	region	model	scenario	
Emissions CO2 Fossil	GtC / yr	World NH	idealised	idealised	0.0
→	6.0		12.0		

(continues on next page)

(continued from previous page)

```

      8.0      World|SH idealised idealised      2.0
      14.0
>>>
>>> afolu = start.filter(variable="*AFOLU")
>>> afolu.head()
time                                2010-01-01 00:00:00
2020-01-01 00:00:00  2030-01-01 00:00:00
variable      unit      region  model      scenario
Emissions|CO2|AFOLU GtC / yr World|NH idealised idealised      1.0
      7.0                                13.0
      World|SH idealised idealised      3.0
      9.0                                15.0
>>>
>>> fos_minus_afolu = fos.subtract(
...     afolu, op_cols={"variable": "Emissions|CO2|Fossil - AFOLU"}
... )
>>> fos_minus_afolu.head()
time                                2010-01-
01 00:00:00  2020-01-01 00:00:00  2030-01-01 00:00:00
model      scenario region  variable      unit
idealised idealised World|NH Emissions|CO2|Fossil - AFOLU gigatC / a
      -1.0              -1.0              -1.0
      World|SH Emissions|CO2|Fossil - AFOLU gigatC / a
      -1.0              -1.0              -1.0
>>>
>>> nh_minus_sh = nh.subtract(sh, op_cols={"region": "World|NH - SH"})
>>> nh_minus_sh.head()
time                                2010-01-01
00:00:00  2020-01-01 00:00:00  2030-01-01 00:00:00
model      scenario region  variable      unit
idealised idealised World|NH - SH Emissions|CO2|Fossil gigatC / a
      -2.0              -2.0              -2.0
      Emissions|CO2|AFOLU gigatC / a
      -2.0              -2.0              -2.0

```

**tail**(\*args: Any, \*\*kwargs: Any) → [pandas.core.frame.DataFrame](#)  
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** [pandas.DataFrame](#)

**time\_mean**(rule: str)

Take time mean of self

Note that this method will not copy the metadata attribute to the returned value.

**Parameters** **rule** ([ "AC", "AS", "A" ]) – How to take the time mean. The names reflect the [pandas user guide](#) where they can, but only the options given above are supported. For clarity, if rule is 'AC', then the mean is an annual mean i.e. each time point in the result is the mean of all values for that particular year. If rule is 'AS', then the mean is an annual mean centred



on the beginning of the year i.e. each time point in the result is the mean of all values from July 1st in the previous year to June 30 in the given year. If rule is 'A', then the mean is an annual mean centred on the end of the year i.e. each time point in the result is the mean of all values from July 1st of the given year to June 30 in the next year.

**Returns** The time mean of `self`.

**Return type** `ScmRun`

#### **property** `time_points`

Time points of the data

**Returns**

**Return type** `scmdata.time.TimePoints`

**timeseries**(*meta=None, check\_duplicated=True, time\_axis=None, drop\_all\_nan\_times=False*)

Return the data with metadata as a `pandas.DataFrame`.

#### **Parameters**

- **meta** (*list[str]*) – The list of meta columns that will be included in the output's Multi-Index. If None (default), then all metadata will be used.
- **check\_duplicated** (*bool*) – If True, an exception is raised if any of the timeseries have duplicated metadata
- **time\_axis** (*{None, "year", "year-month", "days since 1970-01-01", "seconds since 1970-01-01"}*) – See `long_data()` for a description of the options.
- **drop\_all\_nan\_times** (*bool*) – Should time points which contain only nan values be dropped? This operation is applied after any transforms introduced by the value of `time_axis`.

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

**Return type** `pandas.DataFrame`

#### **Raises**

- **NonUniqueMetadataError** – If the metadata are not unique between timeseries and `check_duplicated` is True
- **NotImplementedError** – The value of `time_axis` is not recognised
- **ValueError** – The value of `time_axis` would result in columns which aren't unique

**to\_csv**(*fname: str, \*\*kwargs: Any*) → `None`

Write timeseries data to a csv file

**Parameters** **fname** – Path to write the file into

**to\_iamdataframe**() → `None`

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\_nc**(*fname*, *dimensions*=('region'), *extras*=(), *\*\*kwargs*)

Write timeseries to disk as a netCDF4 file

Each unique variable will be written as a variable within the netCDF file. Choosing the dimensions and extras such that there are as few empty (or nan) values as possible will lead to the best compression on disk.

#### Parameters

- **fname** (*str*) – Path to write the file into
- **dimensions** (*iterable of str*) – Dimensions to include in the netCDF file. The time dimension is always included (if not provided it will be the last dimension). An additional dimension (specifically a co-ordinate in xarray terms), “\_id”, will be included if **extras** is provided and any of the metadata in **extras** is not uniquely defined by **dimensions**. “\_id” maps the timeseries in each variable to their relevant metadata.
- **extras** (*iterable of str*) – Metadata columns to write as variables in the netCDF file (specifically as “non-dimension co-ordinates” in xarray terms, see [xarray terminology](#) for more details). Where possible, these non-dimension co-ordinates will use dimension co-ordinates as their own co-ordinates. However, if the metadata in **extras** is not defined by a single dimension in **dimensions**, then the **extras** co-ordinates will have dimensions of “\_id”. This “\_id” co-ordinate maps the values in the **extras** co-ordinates to each timeseries in the serialised dataset. Where “\_id” is required, an extra “\_id” dimension will also be added to **dimensions**.
- **kwargs** – Passed through to `xarray.Dataset.to_netcdf()`

See also:

`scmdata.run.ScmRun.to_nc()`

**to\_xarray**(*dimensions*=('region'), *extras*=(), *unify\_units*=True)

Convert to a `xarray.Dataset`

#### Parameters

- **dimensions** (*iterable of str*) –  
**Dimensions for each variable in the returned dataset. If an “\_id” co-ordinate is required** (see **extras** documentation for when “\_id” is required) and is not included in **dimensions** then it will be the last dimension (or second last dimension if “time” is also not included in **dimensions**). If “time” is not included in **dimensions** it will be the last dimension.
- **extras** (*iterable of str*) – Columns in `self.meta` from which to create “non-dimension co-ordinates” (see [xarray terminology](#) for more details). These non-dimension co-ordinates store extra information and can be mapped to each timeseries found in the data variables of the output `xarray.Dataset`. Where possible, these non-dimension co-ordinates will use dimension co-ordinates as their own co-ordinates. However, if the metadata in **extras** is not defined by a single dimension in **dimensions**, then the **extras** co-ordinates will have dimensions of “\_id”. This “\_id” co-ordinate maps the values in the **extras** co-ordinates to each timeseries in the serialised dataset. Where “\_id” is required, an extra “\_id” dimension will also be added to **dimensions**.
- **unify\_units** (*bool*) – If a given variable has multiple units, should we attempt to unify them?

**Returns** Data in self, re-formatted as an `xarray.Dataset`

**Return type** `xarray.Dataset`

**Raises**

- **ValueError** – If a variable has multiple units and `unify_units` is `False`.
- **ValueError** – If a variable has multiple units which are not able to be converted to a common unit because they have different base units.

**property values:** `numpy.ndarray`

Timeseries values without metadata

The values are returned such that each row is a different timeseries being a row and each column is a different time (although no time information is included as a plain `numpy.ndarray` is returned).

**Returns** The array in the same shape as `ScmRun.shape()`, that is `(num_timeseries, num_timesteps)`.

**Return type** `np.ndarray`

`scmdata.run.run_append(runs, inplace: bool = False, duplicate_msg: Union[str, bool] = True, metadata: Optional[Dict[str, Union[str, int, float]]] = None)`

Append together many objects.

When appending many objects, it may be more efficient to call this routine once with a list of `ScmRun`'s, than using `ScmRun.append()` multiple times.

If timeseries with duplicate metadata are found, the timeseries are appended and values falling on the same timestep are averaged if `duplicate_msg` is not "return". If `duplicate_msg` is "return", then the result will contain the duplicated timeseries for further inspection.

```
>>> res = base.append(other, duplicate_msg="return")
<scmdata.ScmRun (timeseries: 5, timepoints: 3)>
Time:
  Start: 2005-01-01T00:00:00
  End: 2015-06-12T00:00:00
Meta:
  scenario      variable  model climate_model region  unit
0  a_scenario    Primary Energy  a_iam      a_model    World  EJ/yr
1  a_scenario    Primary Energy|Coal  a_iam      a_model    World  EJ/yr
2  a_scenario2    Primary Energy  a_iam      a_model    World  EJ/yr
3  a_scenario3    Primary Energy  a_iam      a_model    World  EJ/yr
4  a_scenario    Primary Energy  a_iam      a_model    World  EJ/yr
>>> ts = res.timeseries(check_duplicated=False)
>>> ts[ts.index.duplicated(keep=False)]
time                                     2005-01-01  ...  2015-
→ 06-12
scenario  variable      model climate_model region unit      ...
a_scenario Primary Energy a_iam a_model    World  EJ/yr      1.0  ...
→ 7.0
                                     EJ/yr      -1.0  ...
→ 1.0
```

### Parameters

- **runs** (list of `ScmRun`) – The runs to append. Values will be attempted to be cast to `ScmRun`.
- **inplace** – If `True`, then the operation updates the first item in `runs` and returns `None`.
- **duplicate\_msg** – If `True`, raise a `NonUniqueMetadataError` error so the user can see the duplicate timeseries. If `False`, take the average and do not raise a warning or error. If "warn", raise a warning if duplicate data is detected.

- **metadata** – If not None, override the metadata of the resulting *ScmRun* with metadata. Otherwise, the metadata for the runs are merged. In the case where there are duplicate metadata keys, the values from the first run are used.

**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.

**Return type** *ScmRun*

**Raises**

- **TypeError** – If inplace is True but the first element in dfs is not an instance of *ScmRun* runs argument is not a list
- **ValueError** – duplicate\_msg option is not recognised.  
No runs are provided to be appended

### 1.1.14 scmdata.testing

Testing utilities

scmdata.testing.assert\_scm\_df\_almost\_equal(*left, right, allow\_unordered=False, check\_ts\_names=True, rtol=1e-05, atol=1e-08*)

Check that left and right *ScmRun* are equal.

**Parameters**

- **left** (*ScmRun*) –
- **right** (*ScmRun*) –
- **allow\_unordered** (*bool*) – If true, the order of the timeseries is not checked
- **check\_ts\_names** (*bool*) – If true, check that the meta names are the same
- **rtol** (*float*) – Relative tolerance on numeric comparisons
- **atol** (*float*) – Absolute tolerance on numeric comparisons

**Raises** **AssertionError** – left and right are not equal

### 1.1.15 scmdata.time

Time period handling and interpolation

A large portion of this module was originally from openscm. Thanks to the original author, Sven Willner

**exception** scmdata.time.InsufficientDataError

Bases: *Exception*

Insufficient data is available to interpolate/extrapolate

**args**

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**class** scmdata.time.TimePoints(*values*)

Bases: *object*

Handles time points by wrapping *numpy.ndarray* of *numpy.datetime64*..

**as\_cftime**(date\_cls=<class 'cftime.\_cftime.DatetimeGregorian'>) → list

Format time points as cftime.datetime

**Parameters** **date\_cls** (cftime.datetime) – The time points will be returned as instances of date\_cls

**Returns** Time points as a list of date\_cls objects

**Return type** list of cftime.datetime

**days()** → numpy.ndarray

Get day of each time point.

**Returns** Day of each time point

**Return type** numpy.ndarray of int

**hours()** → numpy.ndarray

Get hour of each time point.

**Returns** Hour of each time point

**Return type** numpy.ndarray of int

**months()** → numpy.ndarray

Get month of each time point.

**Returns** Month of each time point

**Return type** numpy.ndarray of int

**to\_index()** → pandas.core.indexes.base.Index

Get time points as pandas.Index.

**Returns** pandas.Index of numpy.dtype object with name "time" made from the time points represented as datetime.datetime.

**Return type** pandas.Index

**property values:** numpy.ndarray

Time points

**weekdays()** → numpy.ndarray

Get weekday of each time point.

**Returns** Day of the week of each time point

**Return type** numpy.ndarray of int

**years()** → numpy.ndarray

Get year of each time point.

**Returns** Year of each time point

**Return type** numpy.ndarray of int

**class** scmdata.time.**TimeseriesConverter**(source\_time\_points: numpy.ndarray, target\_time\_points: numpy.ndarray, interpolation\_type='linear', extrapolation\_type='linear')

Bases: object

Interpolator used to convert data between different time bases

This is a modified version originally in openscm.time.TimeseriesConverter. The integral preserving interpolation was removed as it is outside the scope of this package.

**Parameters**

- **source\_time\_points** (*np.ndarray*) – Source timeseries time points
- **target\_time\_points** (*np.ndarray*) – Target timeseries time points
- **interpolation\_type** (*{"linear"}*) – Interpolation type. Options are 'linear'
- **extrapolation\_type** (*{"linear", "constant", None}*) – Extrapolation type. Options are None, 'linear' or 'constant'

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

**convert\_from**(*values: numpy.ndarray*) → *numpy.ndarray*

Convert value **from** source timeseries time points to target timeseries time points.

**Parameters** **values** (*np.ndarray*) – Value

**Returns** Converted data for timeseries values into the target timebase

**Return type** *np.ndarray*

**convert\_to**(*values: numpy.ndarray*) → *numpy.ndarray*

Convert value from target timeseries time points **to** source timeseries time points.

**Parameters** **values** (*np.ndarray*) – Value

**Returns** Converted data for timeseries values into the source timebase

**Return type** *np.ndarray*

**points\_are\_compatible**(*source: numpy.ndarray, target: numpy.ndarray*) → *bool*

Are the two sets of time points compatible i.e. can I convert between the two?

**Parameters**

- **source** – Source timeseries time points
- **target** – Target timeseries time points

**Returns** Can I convert between the time points?

**Return type** *bool*

## 1.1.16 scmdata.timeseries

TimeSeries handling

Functionality for handling and storing individual time-series

**class** `scmdata.timeseries.TimeSeries`(*data, time=None, \*\*kwargs*)

Bases: `scmdata._base.OpsMixin`

A 1D time-series with metadata

Proxies an `xarray.DataArray` with a single time dimension

**copy**()

Create a deep copy of the timeseries.

Any further modifications to the `Timeseries` returned copy will not be reflected in the current `Timeseries`

**Returns**

**Return type** `Timeseries`

**interpolate**(*target\_times: Union[numpy.ndarray, List[Union[datetime.datetime, int]]], interpolation\_type: str = 'linear', extrapolation\_type: str = 'linear'*)

Interpolate the timeseries onto a new time axis

**Parameters**

- **target\_times** – Time grid onto which to interpolate
- **interpolation\_type** (*str*) – Interpolation type. Options are ‘linear’
- **extrapolation\_type** (*str* or *None*) – Extrapolation type. Options are *None*, ‘linear’ or ‘constant’

**Returns** A new TimeSeries with the new time dimension

**Return type** *TimeSeries*

**property meta**

Metadata associated with the timeseries

**Returns**

**Return type** *dict*

**property name**

Timeseries name

If no name was provided this will be an automatically incrementing number

**reindex**(*time*, *\*\*kwargs*)

Update the time dimension, filling in the missing values with NaN’s

This is different to interpolating to fill in the missing values. Uses *xarray.DataArray.reindex* to perform the reindexing

**Parameters**

- **time** (*obj*:*np.ndarray*) – Time values to reindex the data to. Should be *np.datetime64* values
- **\*\*kwargs** – Additional arguments passed to *xarray*’s *DataArray.reindex* function

**Returns** A new TimeSeries with the new time dimension

**Return type** *TimeSeries*

**References**

[http://xarray.pydata.org/en/stable/generated/xarray.DataArray.reindex\\_like.html#xarray.DataArray.reindex\\_like](http://xarray.pydata.org/en/stable/generated/xarray.DataArray.reindex_like.html#xarray.DataArray.reindex_like)

**property time\_points**

Time points of the data

**Returns**

**Return type** *numpy.ndarray*

**property values**

Get the data as a numpy array

**Returns**

**Return type** *numpy.ndarray*

### 1.1.17 scmdata.units

Unit handling

**class** `scmdata.units.UnitConverter`(*source: str, target: str, context: Optional[str] = None*)

Bases: `object`

Converts numbers between two units.

**property contexts:** `Sequence[str]`

Available contexts for unit conversions

**convert\_from**(*v: Union[float, numpy.ndarray]*) → `Union[float, numpy.ndarray]`

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: Union[float, numpy.ndarray]*) → `Union[float, numpy.ndarray]`

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:** `str`

Source unit

**property target:** `str`

Target unit

**property unit\_registry:** `openscm_units._unit_registry.ScmUnitRegistry`

Underlying unit registry

### 1.1.18 Changelog

#### v0.13.0

- (#174) Add `scmdata.processing.categorisation_sr15()` (also added functionality for this to `scmdata.processing.calculate_summary_stats()`)
- (#173) Add `scmdata.processing.calculate_peak()` and `scmdata.processing.calculate_peak_time()` (also added functionality for these to `scmdata.processing.calculate_summary_stats()`)
- (#175) Remove unused `scmdata.REQUIRED_COLS` (closes #166)
- (#172) Add `scmdata.processing.calculate_summary_stats()`
- (#171) Add `scmdata.processing.calculate_exceedance_probabilities()`, `scmdata.processing.calculate_exceedance_probabilities_over_time()` and `scmdata.ScmRun.get_meta_columns_except()`
- (#170) Added `scmdata.ScmRun.groupby_all_except()` to allow greather use of the concept of grouping by columns except a given set
- (#169) Make `scmdata.processing.calculate_crossing_times()` able to be used as a standalone function rather than being intended to be called via `scmdata.ScmRun.process_over()`



- (#168) Improve the error messages when checking that `scmdata.ScmRun` objects are identical
- (#165) Add `scmdata.processing.calculate_crossing_times()`
- (#164) Added `scmdata.ScmRun.append_timewise()` to allow appending of data along the time axis with broadcasting along multiple meta dimensions
- (#164) Sort time axis internally (ensures that `scmdata.ScmRun.__repr__()` renders properly)
- (#164) Added `scmdata.errors.DuplicateTimesError`, raised when duplicate times are passed to `scmdata.ScmRun`
- (#164) Unified capitalisation of error messages in `scmdata.errors` and added the meta table to `exc_info` of `NonUniqueMetadataError`
- (#163) Added `scmdata.ScmRun.adjust_median_to_target()` to allow for the median of an ensemble of timeseries to be adjusted to a given value
- (#163) Update `scmdata.plotting.RCMIP_SCENARIO_COLOURS` to new AR6 colours

### v0.12.1

- (#162) Fix bug which led to a bad read in when the saved data spanned from before year 1000
- (#162) Allowed `scmdata.ScmRun.plumeplot()` to handle the case where not all data will make complete plumes or have a best-estimate line if `pre_calculated` is `True`. This allows a dataset with one source that has a best-estimate only to be plotted at the same time as a dataset which has a range too with only a single call to `scmdata.ScmRun.plumeplot()`.

### v0.12.0

- (#161) Loosen requirements and drop Python3.6 support

### v0.11.0

- (#159) Allow access to more functions in `scmdata.run.BaseScmRun.process_over`, including arbitrary functions
- (#158) Return `cftime.DatetimeGregorian` rather than `cftime.datetime` from `scmdata.time.TimePoints.as_cftime()` and `scmdata.offsets.generate_range()` to ensure better interoperability with other libraries (e.g. `xarray`'s plotting functionality). Add `date_cls` argument to `scmdata.time.TimePoints.as_cftime()` and `scmdata.offsets.generate_range()` so that the output date type can be user specified.
- (#148) Refactor `scmdata.database.ScmDatabase` to be able to use custom backends
- (#157) Add `disable_tqdm` parameter to `scmdata.database.ScmDatabase.load()` and `scmdata.database.ScmDatabase.save()` to disable displaying progress bars
- (#156) Fix `pandas` and `xarray` documentation links
- (#155) Simplify `flake8` configuration

### v0.10.1

- (#154) Refactor common binary operators for `scmdata.run.BaseScmRun` and `scmdata.timeseries.Timeseries` into a mixin following the removal of `xarray.core.ops.inject_binary_ops()` in *xarray==1.18.0*

### v0.10.0

- (#151) Add `ScmRun.to_xarray()` (improves conversion to xarray and ability of user to control dimensions etc. when writing netCDF files)
- (#149) Fix bug in testcase for `xarray<=0.16.1`
- (#147) Re-do netCDF reading and writing to make use of xarray and provide better handling of extras (results in speedups of 10-100x)
- (#146) Update CI-CD workflow to include more sensible dependencies and also test Python3.9
- (#145) Allow `ScmDatabase.load()` to handle lists as filter values

### v0.9.1

- (#144) Fix `ScmRun.plumeplot()` style handling (previously, if dashes was not supplied each line would be a different style even if all the lines had the same value for `style_var`)

### v0.9.0

- (#143) Alter time axis when serialising to netCDF so that time axis is easily read by other tools (e.g. xarray)

### v0.8.0

- (#139) Update filter to handle metadata columns which contain a mix of data types
- (#139) Add `ScmRun.plumeplot()`
- (#140) Add workaround for installing scmdata with Python 3.6 on windows to handle lack of cftime 1.3.1 wheel
- (#138) Add `ScmRun.quantiles_over()`
- (#137) Fix `scmdata.ScmRun.to_csv()` so that writing and reading is circular (i.e. you end up where you started if you write a file and then read it straight back into a new *scmdata.ScmRun* instance)

### v0.7.6

- (#136) Make filtering by year able to handle a `np.ndarray` of integers (previously this would raise a `TypeError`)
- (#135) Make `scipy` lazy loading in `scmdata.time` follow lazy loading seen in other modules
- (#134) Add CI run in which `seaborn` is not installed to check `scipy` importing

### v0.7.5

- (#133) Pin pandas<1.2 to avoid pint-pandas installation failure (see [pint-pandas #51](#))

### v0.7.4

- (#132) Update to new openscm-units 0.2
- (#130) Add stack info to warning message when filtering results in an empty `scmdata.run.ScmRun`

### v0.7.3

- (#124) Add `scmdata.run.BaseScmRun` and `scmdata.run.BaseScmRun.required_cols` so new sub-classes can be defined which use a different set of required columns from `scmdata.run.ScmRun`. Also added `scmdata.errors.MissingRequiredColumn` and tidied up the docs.
- (#75) Add test to ensure that `scmdata.ScmRun.groupby()` cannot pick up the same timeseries twice even if metadata is changed by the function being applied
- (#125) Fix edge-case when filtering an empty `scmdata.ScmRun`
- (#123) Add `scmdata.database.ScmDatabase` to read/write data using multiple files. (closes #103)

### v0.7.2

- (#121) Faster implementation of `scmdata.run.run_append()`. The original timeseries indexes and order are no longer maintained after an append.
- (#120) Check the type and length of the runs argument in `scmdata.run.run_append()` (closes #101)

### v0.7.1

- (#119) Make groupby support grouping by metadata with integer values
- (#119) Ensure using `scmdata.run.run_append()` does not mangle the index to `pd.DatetimeIndex`

### v0.7.0

- (#118) Make scipy an optional dependency
- (#117) Sort timeseries index ordering (closes #97)
- (#116) Update `scmdata.ScmRun.drop_meta()` inplace behaviour
- (#115) Add `na_override` argument to `scmdata.ScmRun.process_over()` for handling nan metadata (closes #113)
- (#114) Add operations: `scmdata.ScmRun.linear_regression()`, `scmdata.ScmRun.linear_regression_gradient()`, `scmdata.ScmRun.linear_regression_intercept()` and `scmdata.ScmRun.linear_regression_scmrun()`
- (#111) Add operation: `scmdata.ScmRun.delta_per_delta_time()`
- (#112) Ensure unit conversion doesn't fall over when the target unit is in the input
- (#110) Revert to using `pd.DataFrame` with `pd.Categorical` series as meta indexes.

- (#108) Remove deprecated `ScmDataFrame` (closes #60)
- (#105) Add performance benchmarks for `ScmRun`
- (#106) Add `ScmRun.integrate()` so we can integrate timeseries with respect to time
- (#104) Fix bug when reading csv/excel files which use integer years and `lowercase_cols=True` (closes #102)

#### **v0.6.4**

- (#96) Fix non-unique timeseries metadata checks for `ScmRun.timeseries()`
- (#100) When initialising `ScmRun` from file, make the default be to read with `pd.read_csv()`. This means we now initialising reading from gzipped CSV files.
- (#99) Hotfix failing notebook test
- (#94) Fix edge-case issue with `drop_meta` (closes #92)
- (#95) Add `drop_all_nan_times` keyword argument to `ScmRun.timeseries()` so time points with no data of interest can easily be removed

#### **v0.6.3**

- (#91) Provide support for `pandas==1.1`

#### **v0.6.2**

- (#87) Upgrade workflow to use `isort>=5`
- (#82) Add support for adding Pint scalars and vectors to `scmdata.Timeseries` and `scmdata.ScmRun` instances
- (#85) Allow required columns to be read as `extras` from netCDF files (closes #83)
- (#84) Raise a `DeprecationWarning` if no default `inplace` argument is provided for `ScmRun.drop_meta()`. `inplace` default behaviour scheduled to be changed to `False` in v0.7.0
- (#81) Add `scmdata.run.ScmRun.metadata` to track `ScmRun` instance-specific metadata (closes #77)
- (#80) No longer use `pandas.tseries.offsets.BusinessMixin` to determine Business-related offsets in `scmdata.offsets.to_offset()`. (closes #78)
- (#79) Introduce `scmdata.errors.NonUniqueMetadataError`. Update handling of duplicate metadata so default behaviour of `run_append` is to raise a `NonUniqueMetadataError`. (closes #76)

#### **v0.6.1**

- (#74) Update handling of unit conversion context during unit conversions
- (#73) Only `reindex` timeseries when dealing with different time points

## v0.5.2

- (#65) Use pint for ops, making them automatically unit aware
- (#71) Start adding arithmetic support via `scmdata.ops`. So far only add and subtract are supported.
- (#70) Automatically set y-axis label to units if it makes sense in `ScmRun`'s `lineplot()` method

## v0.5.1

- (#68) Rename `scmdata.run.df_append()` to `:func`scmdata.run.run_append``. `:func`scmdata.run.df_append`` deprecated and will be removed in v0.6.0
- (#67) Update the documentation for `ScmRun.append()`
- (#66) Raise `ValueError` if index/columns arguments are not provided when instantiating a `:class`ScmRun`` object with a numpy array. Add `lowercase_cols` argument to coerce the column names in CSV files to lowercase

## v0.5.0

- (#64) Remove spurious warning from `ScmRun`'s `filter()` method
- (#63) Remove `set_meta()` from `ScmRun` in preference for using the `__setitem__()` method
- (#62) Fix interpolation when the data contains nan values
- (#61) Hotfix filters to also include caret ("`^`") in pseudo-regex syntax. Also adds `empty()` property to `ScmRun`
- (#59) Deprecate `ScmDataFrame`. To be removed in v0.6.0
- (#58) Use `cftime` datetimes when appending `ScmRun` objects to avoid `OutOfBounds` errors when datetimes span many centuries
- (#55) Add `time_axis` keyword argument to `ScmRun.timeseries`, `ScmRun.long_data` and `ScmRun.lineplot` to give greater control of the time axis when retrieving data
- (#54) Add `drop_meta()` to `ScmRun` for dropping metadata columns
- (#53) Don't convert case of variable names written to file. No longer convert case of serialized dataframes
- (#51) Refactor `relative_to_ref_period_mean()` so that it returns an instance of the input data type (rather than a `pd.DataFrame`) and puts the reference period in separate meta columns rather than mangling the variable name.
- (#47) Update README and `setup.py` to make it easier for new users

## v0.4.3

- (#46) Add test of conda installation

**v0.4.2**

- (#45) Make installing seaborn optional

**v0.4.1**

- (#44) Add multi-dimensional handling to `scmdata.netcdf`
- (#43) Fix minor bugs in netCDF handling and address minor code coverage issues
- (#41) Update documentation of the data model. Additionally:
  - makes `.time_points` attributes consistently return `scmdata.time.TimePoints` instances
  - ensures `.meta` is used consistently throughout the code base (removing `.metadata`)
- (#33) Remove dependency on `pyam`. Plotting is done with `seaborn` instead.
- (#34) Allow the serialization/deserialization of `scmdata.run.ScmRun` and `scmdata.ScmDataFrame` as netCDF4 files.
- (#30) Swap to using `openscm-units` for unit handling (hence remove much of the `scmdata.units` module)
- (#21) Added `scmdata.run.ScmRun` as a proposed replacement for `scmdata.dataframe.ScmDataFrame`. This new class provides an identical interface as a `ScmDataFrame`, but uses a different underlying data structure to the `ScmDataFrame`. The purpose of `ScmRun` is to provide performance improvements when handling large sets of time-series data. Removed support for Python 3.5 until `pyam` dependency is optional
- (#31) Tidy up repository after changing location

**v0.4.0**

- (#28) Expose `scmdata.units.unit_registry`

**v0.3.1**

- (#25) Make `scipy` an optional dependency
- (#24) Fix missing “N2O” unit (see #14). Also updates test of year to day conversion, it is 365.25 to within 0.01% (but depends on the Pint release).

**v0.3.0**

- (#20) Add support for `python=3.5`
- (#19) Add support for `python=3.6`

### v0.2.2

- (#16) Only rename columns when initialising data if needed

### v0.2.1

- (#13) Ensure LICENSE is included in package
- (#11) Add SO2F2 unit and update to Pyam v0.3.0
- (#12) Add `get_unique_meta` convenience method
- (#10) Fix extrapolation bug which prevented any extrapolation from occurring

### v0.2.0

- (#9) Add `time_mean` method
- (#8) Add `make docs` target

### v0.1.2

- (#7) Add notebook tests
- (#4) Unit conversions for CH4 and N2O contexts now work for compound units (e.g. 'Mt CH4 / yr' to 'Gt C / day')
- (#6) Add auto-formatting

### v0.1.1

- (#5) Add `scmdata.dataframe.df_append` to `__init__.py`

### v0.1.0

- (#3) Added documentation for the api and Makefile targets for releasing
- (#2) Refactored `scmdataframe` from [openclimatedata/openscm@077f9b5](https://github.com/openclimatedata/openscm) into a standalone package
- (#1) Add docs folder





## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### S

- `scmdata.database`, 8
- `scmdata.errors`, 11
- `scmdata.filters`, 11
- `scmdata.groupby`, 14
- `scmdata.netcdf`, 19
- `scmdata.offsets`, 20
- `scmdata.ops`, 22
- `scmdata.plotting`, 32
- `scmdata.processing`, 34
- `scmdata.testing`, 64
- `scmdata.time`, 64
- `scmdata.timeseries`, 66
- `scmdata.units`, 68



## Symbols

`__init__()` (*scmdata.run.ScmRun* method), 38

## A

`add()` (in module *scmdata.ops*), 22

`add()` (*scmdata.run.ScmRun* method), 39

`adjust_median_to_target()` (in module *scmdata.ops*), 24

`adjust_median_to_target()` (*scmdata.run.ScmRun* method), 42

`all()` (*scmdata.groupby.RunGroupBy* method), 14

`any()` (*scmdata.groupby.RunGroupBy* method), 14

`append()` (*scmdata.run.ScmRun* method), 42

`append_timewise()` (*scmdata.run.ScmRun* method), 43

`args` (*scmdata.time.InsufficientDataError* attribute), 64

`as_cftime()` (*scmdata.time.TimePoints* method), 64

`assert_scm_df_almost_equal()` (in module *scmdata.testing*), 64

`available_data()` (*scmdata.database.ScmDatabase* method), 9

## C

`calculate_crossing_times()` (in module *scmdata.processing*), 34

`calculate_exceedance_probabilities()` (in module *scmdata.processing*), 35

`calculate_exceedance_probabilities_over_time()` (in module *scmdata.processing*), 35

`calculate_peak()` (in module *scmdata.processing*), 36

`calculate_peak_time()` (in module *scmdata.processing*), 36

`calculate_summary_stats()` (in module *scmdata.processing*), 36

`categorisation_sr15()` (in module *scmdata.processing*), 37

`contexts` (*scmdata.units.UnitConverter* property), 68

`convert_from()` (*scmdata.time.TimeseriesConverter* method), 66

`convert_from()` (*scmdata.units.UnitConverter* method), 68

`convert_to()` (*scmdata.time.TimeseriesConverter* method), 66

`convert_to()` (*scmdata.units.UnitConverter* method), 68

`convert_unit()` (*scmdata.run.ScmRun* method), 43

`copy()` (*scmdata.run.ScmRun* method), 43

`copy()` (*scmdata.timeseries.TimeSeries* method), 66

`count()` (*scmdata.groupby.RunGroupBy* method), 14

## D

`data_hierarchy_separator` (*scmdata.run.ScmRun* attribute), 43

`DatabaseBackend` (class in *scmdata.database*), 8

`datetime_match()` (in module *scmdata.filters*), 11

`day_match()` (in module *scmdata.filters*), 11

`days()` (*scmdata.time.TimePoints* method), 65

`delete()` (*scmdata.database.DatabaseBackend* method), 8

`delete()` (*scmdata.database.NetCDFBackend* method), 8

`delete()` (*scmdata.database.ScmDatabase* method), 10

`delta_per_delta_time()` (in module *scmdata.ops*), 25

`delta_per_delta_time()` (*scmdata.run.ScmRun* method), 43

`divide()` (in module *scmdata.ops*), 25

`divide()` (*scmdata.run.ScmRun* method), 44

`drop_meta()` (*scmdata.run.ScmRun* method), 46

`DuplicateTimesError`, 11

## E

`empty` (*scmdata.run.ScmRun* property), 46

`ensure_dir_exists()` (in module *scmdata.database*), 10

## F

`filter()` (*scmdata.run.ScmRun* method), 46

`find_depth()` (in module *scmdata.filters*), 11

`from_nc()` (*scmdata.run.ScmRun* class method), 47

## G

`generate_range()` (in module *scmdata.offsets*), 20

`get()` (*scmdata.database.DatabaseBackend* method), 8

`get()` (*scmdata.database.NetCDFBackend* method), 8

`get_key()` (*scmdata.database.NetCDFBackend method*), 9  
`get_meta_columns_except()` (*scmdata.run.ScmRun method*), 48  
`get_unique_meta()` (*scmdata.run.ScmRun method*), 48  
`groupby()` (*scmdata.run.ScmRun method*), 48  
`groupby_all_except()` (*scmdata.run.ScmRun method*), 49

## H

`head()` (*scmdata.run.ScmRun method*), 49  
`hour_match()` (*in module scmdata.filters*), 12  
`hours()` (*scmdata.time.TimePoints method*), 65

## I

`inject_nc_methods()` (*in module scmdata.netcdf*), 19  
`inject_ops_methods()` (*in module scmdata.ops*), 27  
`inject_plotting_methods()` (*in module scmdata.plotting*), 32  
`InsufficientDataError`, 64  
`integrate()` (*in module scmdata.ops*), 27  
`integrate()` (*scmdata.run.ScmRun method*), 49  
`interpolate()` (*scmdata.run.ScmRun method*), 49  
`interpolate()` (*scmdata.timeseries.TimeSeries method*), 66  
`is_in()` (*in module scmdata.filters*), 12

## L

`line_plot()` (*scmdata.run.ScmRun method*), 49  
`linear_regression()` (*in module scmdata.ops*), 27  
`linear_regression()` (*scmdata.run.ScmRun method*), 49  
`linear_regression_gradient()` (*in module scmdata.ops*), 27  
`linear_regression_gradient()` (*scmdata.run.ScmRun method*), 50  
`linear_regression_intercept()` (*in module scmdata.ops*), 27  
`linear_regression_intercept()` (*scmdata.run.ScmRun method*), 50  
`linear_regression_scmrun()` (*in module scmdata.ops*), 28  
`linear_regression_scmrun()` (*scmdata.run.ScmRun method*), 50  
`lineplot()` (*in module scmdata.plotting*), 32  
`lineplot()` (*scmdata.run.ScmRun method*), 50  
`load()` (*scmdata.database.DatabaseBackend method*), 8  
`load()` (*scmdata.database.NetCDFBackend method*), 9  
`load()` (*scmdata.database.ScmDatabase method*), 10  
`long_data()` (*scmdata.run.ScmRun method*), 51

## M

`map()` (*scmdata.groupby.RunGroupBy method*), 15

`max()` (*scmdata.groupby.RunGroupBy method*), 15  
`mean()` (*scmdata.groupby.RunGroupBy method*), 16  
`median()` (*scmdata.groupby.RunGroupBy method*), 16  
`meta` (*scmdata.run.ScmRun property*), 51  
`meta` (*scmdata.timeseries.TimeSeries property*), 67  
`meta_attributes` (*scmdata.run.ScmRun property*), 51  
`min()` (*scmdata.groupby.RunGroupBy method*), 17  
`MissingRequiredColumnError`, 11  
`module`

- `scmdata.database`, 8
- `scmdata.errors`, 11
- `scmdata.filters`, 11
- `scmdata.groupby`, 14
- `scmdata.netcdf`, 19
- `scmdata.offsets`, 20
- `scmdata.ops`, 22
- `scmdata.plotting`, 32
- `scmdata.processing`, 34
- `scmdata.testing`, 64
- `scmdata.time`, 64
- `scmdata.timeseries`, 66
- `scmdata.units`, 68

`month_match()` (*in module scmdata.filters*), 12  
`months()` (*scmdata.time.TimePoints method*), 65  
`multiply()` (*in module scmdata.ops*), 28  
`multiply()` (*scmdata.run.ScmRun method*), 51

## N

`name` (*scmdata.timeseries.TimeSeries property*), 67  
`nc_to_run()` (*in module scmdata.netcdf*), 19  
`NetCDFBackend` (*class in scmdata.database*), 8  
`NonUniqueMetadataError`, 11

## P

`pattern_match()` (*in module scmdata.filters*), 12  
`plumeplot()` (*in module scmdata.plotting*), 33  
`plumeplot()` (*scmdata.run.ScmRun method*), 53  
`points_are_compatible()` (*scmdata.time.TimeseriesConverter method*), 66  
`prep_for_op()` (*in module scmdata.ops*), 29  
`process_over()` (*scmdata.run.ScmRun method*), 54  
`prod()` (*scmdata.groupby.RunGroupBy method*), 17

## Q

`quantiles_over()` (*scmdata.run.ScmRun method*), 55

## R

`reduce()` (*scmdata.groupby.RunGroupBy method*), 17  
`reduce()` (*scmdata.run.ScmRun method*), 55  
`reindex()` (*scmdata.timeseries.TimeSeries method*), 67  
`relative_to_ref_period_mean()` (*scmdata.run.ScmRun method*), 56

required\_cols (*scmdata.run.ScmRun* attribute), 56  
 resample() (*scmdata.run.ScmRun* method), 56  
 root\_dir (*scmdata.database.ScmDatabase* property), 10  
 run\_append() (in module *scmdata.run*), 63  
 run\_to\_nc() (in module *scmdata.netcdf*), 19  
 RunGroupBy (class in *scmdata.groupby*), 14

## S

save() (*scmdata.database.DatabaseBackend* method), 8  
 save() (*scmdata.database.NetCDFBackend* method), 9  
 save() (*scmdata.database.ScmDatabase* method), 10  
 scmdata.database  
   module, 8  
 scmdata.errors  
   module, 11  
 scmdata.filters  
   module, 11  
 scmdata.groupby  
   module, 14  
 scmdata.netcdf  
   module, 19  
 scmdata.offsets  
   module, 20  
 scmdata.ops  
   module, 22  
 scmdata.plotting  
   module, 32  
 scmdata.processing  
   module, 34  
 scmdata.testing  
   module, 64  
 scmdata.time  
   module, 64  
 scmdata.timeseries  
   module, 66  
 scmdata.units  
   module, 68  
 ScmDatabase (class in *scmdata.database*), 9  
 ScmRun (class in *scmdata.run*), 38  
 set\_op\_values() (in module *scmdata.ops*), 30  
 shape (*scmdata.run.ScmRun* property), 58  
 source (*scmdata.units.UnitConverter* property), 68  
 std() (*scmdata.groupby.RunGroupBy* method), 18  
 subtract() (in module *scmdata.ops*), 30  
 subtract() (*scmdata.run.ScmRun* method), 58  
 sum() (*scmdata.groupby.RunGroupBy* method), 18

## T

tail() (*scmdata.run.ScmRun* method), 60  
 target (*scmdata.units.UnitConverter* property), 68  
 time\_match() (in module *scmdata.filters*), 13  
 time\_mean() (*scmdata.run.ScmRun* method), 60  
 time\_points (*scmdata.run.ScmRun* property), 61

time\_points (*scmdata.timeseries.TimeSeries* property), 67  
 TimePoints (class in *scmdata.time*), 64  
 TimeSeries (class in *scmdata.timeseries*), 66  
 timeseries() (*scmdata.run.ScmRun* method), 61  
 TimeseriesConverter (class in *scmdata.time*), 65  
 to\_csv() (*scmdata.run.ScmRun* method), 61  
 to\_iamdataframe() (*scmdata.run.ScmRun* method), 61  
 to\_index() (*scmdata.time.TimePoints* method), 65  
 to\_nc() (*scmdata.run.ScmRun* method), 61  
 to\_xarray() (*scmdata.run.ScmRun* method), 62

## U

unit\_registry (*scmdata.units.UnitConverter* property), 68  
 UnitConverter (class in *scmdata.units*), 68

## V

values (*scmdata.run.ScmRun* property), 63  
 values (*scmdata.time.TimePoints* property), 65  
 values (*scmdata.timeseries.TimeSeries* property), 67  
 var() (*scmdata.groupby.RunGroupBy* method), 19

## W

weekdays() (*scmdata.time.TimePoints* method), 65  
 with\_traceback() (*scmdata.time.InsufficientDataError* method), 64

## Y

years() (*scmdata.time.TimePoints* method), 65  
 years\_match() (in module *scmdata.filters*), 13