xscen Documentation

Release 0.7.25-beta

Gabriel Rondeau-Genesse

Jan 16, 2024

CONTENTS:

xscen: A climate change scenario-building analysis framework, built with Intake-esm catalogs and xarray-based packages such as xclim and xESMF.

CHAPTER

ONE

NEED HELP?

- Ouranos employees can ask questions on the Ouranos private StackOverflow where you can tag subjects and people. (https://stackoverflow.com/c/ouranos/questions).
- Potential bugs in xscen can be reported as an issue here: https://github.com/Ouranosinc/xscen/issues .
- Problems with data on Ouranos' servers can be reported as an issue here: https://github.com/Ouranosinc/miranda/issues
- To be aware of changes in xscen, you can "watch" the github repo. You can customize the watch function to notify you of new releases. (https://github.com/Ouranosinc/xscen)

CHAPTER

TWO

FEATURES

- Supports workflows with YAML configuration files for better transparency, reproducibility, and long-term backups.
- Intake_esm-based catalog to find and manage climate data.
- Climate dataset extraction, subsetting, and temporal aggregation.
- Calculate missing variables through Intake-esm's DerivedVariableRegistry.
- Regridding with xESMF.
- Bias adjustment with xclim.

2.1 xscen

A climate change scenario-building analysis framework, built with Intake-esm catalogs and xarray-based packages such as xclim and xESMF.

For documentation concerning *xscen*, see: https://xscen.readthedocs.io/en/latest/

2.1.1 Features

- Supports workflows with YAML configuration files for better transparency, reproducibility, and long-term backups.
- Intake-esm-based catalog to find and manage climate data.
- Climate dataset extraction, subsetting, and temporal aggregation.
- Calculate missing variables through intake-esm's DerivedVariableRegistry.
- Regridding powered by xESMF.
- Bias adjustment tools provided by xclim.

2.1.2 Installation

Please refer to the installation docs.

2.1.3 Acknowledgments

This package was created with Cookiecutter and the Ouranosinc/cookiecutter-pypackage project template.

2.2 Installation

2.2.1 Official Sources

Because of some packages being absent from PyPI (such as *xESMF*), we strongly recommend installing *xscen* in an Anaconda Python environment.

xscen can be installed directly from conda-forge:

```
$ conda install -c conda-forge xscen
```

Note: If you are unable to install the package due to missing dependencies, ensure that *conda-forge* is listed as a source in your *conda* configuration: *$ conda config –add channels conda-forge*!

If for some reason you wish to install the *PyPI* version of *xscen* into an existing Anaconda environment (*not recommended*), this can be performed with:

```
$ python -m pip install xscen
```

2.2.2 Development Installation (Anaconda + pip)

For development purposes, we provide the means for generating a conda environment with the latest dependencies in an *environment.yml* file at the top-level of the Github repo.

In order to get started, first clone the repo locally:

```
$ git clone git@github.com:Ouranosinc/xscen.git
```

Then you can create the environment and install the package:

```
$ cd xscen
$ conda env create -f environment.yml
```

Finally, perform an *–editable* install of xscen and compile the translation catalogs:

```
$ python -m pip install -e .
$ make translate
```

2.3 Good to know

2.3.1 Which function to use when opening data

There are many ways to open data in xscen workflows. The list below tries to make the differences clear:

Search and extract
: Using *search_data_catalogs()* + *extract_dataset()*. This is the main method recommended to parse catalogs of "raw" data, data not yet modified by your workflow. It has features meant to ease the aggregation and extraction of raw files :

  - variable conversion and resampling of subdaily data
  - spatial and temporal subsetting
  - matching historical and future runs for simulations

  `search_data_catalogs` returns a dictionary with a specific catalog for each of the unique `id` found in the search. One should then iterate over this dictionary and call `extract_dataset` on each item. This then returns a dictionary with a single dataset for each `xrfreq`. You thus end up with one dataset per frequency and `id`.

to_dataset_dict
: Using `to_dataset_dict()`. When all the data you need is in a single catalog (for example, your *ProjectCatalog()*) and you don't need any of the features listed above. Note that this can be combined to a simple *.search* beforehand, to subset on parts of the catalog. As explained in *Columns*, it creates a dictionary with a single Dataset for each combination of `id`, `domain`, `processing_level` and `xrfreq` unless different aggregation rules were called during the catalog creation.

to_dataset
: Using *to_dataset()*. Similar to `to_dataset_dict`, but only returns a single dataset. If the catalog has more than one, the call will fail. It behaves like `to_dask()`, but exposes options to add aggregations. This is useful when constructing an ensemble dataset that would otherwise result in distinct entries in the output of `to_dataset_dict`. It can usually be used in replacement of a combination of `to_dataset_dict` and `create_ensemble()`.

open_dataset
: Of course, xscen workflows can still use the conventional `open_dataset()`. Just be aware that datasets opened this way will lack the attributes automatically added by the previous functions, which will then result in poorer metadata or even failure for some *xscen* functions. Same thing for `open_mfdataset()`. If one has data listed in a catalog, the functions above will usually provide what you need, i.e. : `xr.open_mfdataset(cat.df.path)` is very rarely optimal.

create_ensemble
: With *to_dataset()* or *ensemble_stats()*, you should usually find what you need. `create_ensemble()` is not needed in xscen workflows.

2.3.2 Which function to use when resampling data

extract_dataset
: *extract_dataset()*'s resampling capabilities are meant to provide daily data from finer sources.

resample
: :py:func`xscen.extract.resample` extends xarray's *resample* methods with support for weighted resampling when starting from data coarser than daily and for handling of missing timesteps or values.

xclim indicators
: Through *compute_indicators()*, xscen workflows can easily use xclim indicators to go from daily data to coarser (monthly, seasonal, annual), with missing values handling. This option will add more metadata than the two firsts.

2.3.3 Metadata translation

xscen itself does not add many translatable attributes, but when it does, it will look into xclim's options for which locales to translate them to. Similar to xclim, it will always add a particular attribute in english and then translations with the same attribute name suffixed by "_XX", where "XX" is the two-letter language code, as set in the ISO-639-1 standard. For example, if a function adds a *long_name* and Inuktitut translation is activated, the function will also add a *long_name_iu* attribute.

In a config file, activating French translations for both xclim's indicators and xscen (and figanos) is done with :

```
xclim:
    metadata_locales:
        - fr
```

Which can also be activated in the code using `xclim.core.options.set_options()`. Note that this only applies to attributes that are *added* to a dataset. Some xscen functions will instead update an existing attribute. For example, when calculating the climatology of a variable with *long_name Mean temperature*, `climatological_mean()` will update the *long_name* as *30-year average of Mean temperature*. This automatic update is done for all locales available in the variable, no matter what xclim option is activated. For example, if a *long_name_eu* exists in the variable and a Basque translation catalog exists in that xscen instance, then the attribute will be translated, no matter what xclim's `metadata_locales` is set to.

Translation is of course not automatic but relies on manually populated gettext catalogs. xscen ships with a catalog of french (fr) translations. See *Translating xscen* to learn how to add translations to xscen. xclim's documentation of the same subject is here.

If your xscen is installed in "editable" mode in its source directory (`pip install -e .`), you should run `make translate` each time you pull changes from the upstream source.

2.3.4 Module-wide options

As seen above, it can be useful to use the "special" sections of the config file to set some module-wide options. For example:

```
logging:
    # same arguments as python's logging.config.dictConfig
xarray:
    keep_attrs: True
xclim:
    metadata_locales:
```

(continues on next page)

(continued from previous page)

```
        - fr
    check_missing: "skip"
warning:
    # warning_category : filter_action
    all: ignore
```

2.3.5 Global warming dataset

The *xscen.extract.get_warming_level()* and *xscen.extract.subset_warming_level()* functions use a custom made database of global temperature averages to find the global warming levels of known climate simulations. The database is stored as a netCDF file inside the package itself. It stores the global temperature average (land and ocean) from 1850 to 2100 for multiple simulations (not all simulations cover the entire temporal range). Simulations are defined through 4 fields:

- `mip_era` : "CMIP6", "CMIP5" or "obs" (see below)
- `source` : The model name for GCM (same as the *source* column) and the driving model name for RCM (*driving_model* column)
- `experiment` : The CMIP experiment name of the run. The "historical" and "pre-industrial" experiments have been merged into each future experiment (similar to what `match_hist_and_fut` does in `search_data_catalogs()`)
- `member` : The realization variant label of the run (same as the *member* column)

An extra `data_source` field is also available and describes how the data has been obtained:

- "IPCC Atlas" : The timeseries was copied directly from the public data of the IPCC Atlas'
- "From Amon" : The monthly temperature average was resampled annually and averaged over the globe using a cos-lat weighting
- "From Amon with xscen" : Same, xscen was used to perform the computation.

In addition to the climate simulations, a few "observational" datasets are made available in the database. The choice of datasets and the methodology was adapted from the WMO's State of the Global Climate 2021. However, to have some consistency between these and the simulated series, an estimated 1850-1900 mean temperature was added to the WMO-compliant anomalies to get absolute values. Keep in mind that this is only an estimation, the timeseries should only be used to compute anomalies. The observational series have a short dataset name in the `source` field, "obs" in `mip_era` and `experiment`, and an empty `member` (""). The `data_source` is noted : "Computed following WMO guidelines".

2.4 Examples

2.4.1 Using and understanding Catalogs

INFO

Catalogs in `xscen` are built upon Datastores in `intake_esm`. For more information on basic usage, such as the `search()` function, please consult their documentation.

Catalogs are made of two files:

- JSON file containing metadata such as the catalog's title, description etc. It also contains an attribute *catalog_file* that points towards the CSV. Most xscen catalog will have very similar JSON files.
- CSV file containing the catalog itself. This file can be zipped.

Two types of catalogs have been implemented in `xscen`.

- **Static catalogs:** A `DataCatalog <../xscen.rst#xscen.catalog.DataCatalog>`__ is a *read-only* `intake-esm` catalog that contains information on all available data. Usually, this type of catalog should only be consulted at the start of a new project.
- **Updatable catalogs:** A `ProjectCatalog <../xscen.rst#xscen.catalog.ProjectCatalog>`__ is a *DataCatalog* with additional *write* functionalities. This kind of catalog should be used to keep track of the new data created during the course of a project, such as regridded or bias-corrected data, since it can `update` itself and append new information to the associated CSV file.

NOTE: As to not accidentaly lose data, both catalogs currently have no function to remove data from the CSV file. However, upon initialisation and when updating or refreshing itself, the catalog validates that all entries still exist and, if files have been manually removed, deletes their entries from the catalog.

Catalogs in `xscen` are made to follow a nomenclature that is as close as possible to the Python Earth Science Standard Vocabulary : https://github.com/ES-DOC/pyessv. The columns are listed below but for more details and concrete examples about the entries, consult *the relevant page in the documentation*:

Column name	Description
id	Unique DatasetID generated by `xscen` based on a subset of columns.
type	Type of data: [forecast, station-obs, gridded-obs, reconstruction, simulation]
processing_level	Level of post-processing reached: [raw, extracted, regridded, biasadjusted]
bias_adjust_institution	Institution that computed the bias adjustment.
bias_adjust_project	Name of the project that computed the bias adjustment.
mip_era	CMIP Generation associated with the data.
activity	Model Intercomparison Project (MIP) associated with the data.
driving_model	Name of the driver.
institution	Institution associated with the source.
source	Name of the model or the dataset.
experiment	Name of the experiment of the model.
member	Name of the realisation (or of the driving realisation in the case of RCMs).
xrfreq	Pandas/xarray frequency.
frequency	Frequency in letters (CMIP6 format).
variable	Variable(s) in the dataset.
domain	Name of the region covered by the dataset.
date_start	First date of the dataset.
date_end	Last date of the dataset.
version	Version of the dataset.
format	Format of the dataset.
path	Path to the dataset.

Individual projects may use a different set of columns, but those will always be present in the official Ouranos internal catalogs. Some parts of `xscen` will however expect certain column names, so diverging from the official list is to be done with care.

Basic Catalog Usage

If an official catalog already exists, it should be opened using `xs.DataCatalog` by pointing it to the JSON file:

[ ]:
```
from pathlib import Path

from xscen import DataCatalog, ProjectCatalog

# Prepare a dummy folder where data will be put
output_folder = Path().absolute() / "_data"
output_folder.mkdir(exist_ok=True)

DC = DataCatalog(f"{Path().absolute()}/samples/pangeo-cmip6.json")
DC
```

The content of the catalog can be accessed by a call to `df`, which will return a `pandas.DataFrame`.

[ ]:
```
# Access the catalog
DC.df[0:3]
```

The `unique` function allows listing unique elements for either all the catalog or a subset of columns. It can be called in a few various ways, listed below:

[ ]:
```
# List all unique elements in the catalog, returns a pandas.Series
DC.unique()
```

[ ]:
```
# List all unique elements in a subset of columns, returns a pandas.Series
DC.unique(["variable", "frequency"])
```

[ ]:
```
# List all unique elements in a single columns, returns a list
DC.unique("id")[0:5]
```

Basic .search() commands

The `search` function comes from `intake-esm` and allows searching for specific elements in the catalog's columns. It accepts both wildcards and regular expressions (except for *variable*, which must be exact due to being in *tuples*).

While regex isn't great at inverse matching ("does not contain"), it is possible. Here are a few useful commands:

```
- ^string            : Starts with string

- string$            : Ends with string

- ^(?!string).*$     : Does not start with string

- .*(?<!string)$     : Does not end with string

- ^((?!string).)*$   : Does not contain substring

- ^(?!string$).*$    : Is not that exact string
```

This website can be used to test regex commands: https://regex101.com/

```
[ ]: # Regex: Find all entries that start with "ssp"
print(DC.search(experiment="^ssp").unique("experiment"))
```

```
[ ]: # Regex: Exclude all entries that start with "ssp"
print(DC.search(experiment="^(?!ssp).*$").unique("experiment"))
```

```
[ ]: # Regex: Find all experiments except the exact string "ssp126"
print(DC.search(experiment="^(?!ssp126$).*$").unique("experiment"))
```

```
[ ]: # Wildcard: Find all entries that start with NorESM2
print(DC.search(source="NorESM2.*").unique("source"))
```

Notice that the search function returns everything available that matches some of the criteria.

```
[ ]: # r1i1p1f1 sftlf is not available
DC.search(
    source="NorESM2-MM",
    experiment="historical",
    member=["r1i1p1f1", "r2i1p1f1"],
    variable=["sftlf", "pr"],
).df
```

You can restrict your search to only keep entries that matches all the criteria across a list of columns.

```
[ ]: # Only returns variables that have all members, source and experiment asked for. In this␣
↪case, pr, but not sftlf.
DC.search(
    source="NorESM2-MM",
    experiment="historical",
    member=["r1i1p1f1", "r2i1p1f1"],
    variable=["sftlf", "pr"],
    require_all_on=["variable"],
).df
```

It is also possible to search for files that intersect a specific time period.

```
[ ]: DC.search(periods=[["2016", "2017"]]).unique(["date_start", "date_end"])
```

Advanced search: xs.search_data_catalogs

`search` has multiple notable limitations for more advanced searches:

- It can't match specific criteria together, such as finding a dataset that would have both 3h precipitation and daily temperature.
- It has no explicit understanding of climate datasets, and thus can't match historial and future simulations together or know how realization members or grid resolutions work.

`xs.search_data_catalogs` was thus created as a more advanced version that is closer to the needs of climate services. It also plays the double role of preparing certain arguments for the extraction function.

Due to how different reference datasets are from climate simulations, this function might have to be called multiple times and the results concatenated into a single dictionary. The main arguments are:

- `variables_and_freqs` is used to indicate which variable and which frequency is required. NOTE: With the exception of fixed fields, where *'fx'* should be used, frequencies here use the `pandas` nomenclature ('D', 'H', '6H', 'MS', etc.).
- `other_search_criteria` is used to search for specific entries in other columns of the catalog, such as *activity*. `require_all_on` can also be passed here.
- `exclusions` is used to exclude certain simulations or keywords from the results.
- `match_hist_and_fut` is used to indicate that RCP/SSP simulations should be matched with their *historical* counterparts.
- `periods` is used to search for specific time periods.
- `allow_resampling` is used to allow searching for data at higher frequencies than requested.
- `allow_conversion` is used to allow searching for calculable variables, in the case where the requested variable would not be available.
- `restrict_resolution` is used to limit the results to the finest or coarsest resolution available for each source.
- `restrict_members` is used to limit the results to a maximum number of realizations for each source.
- `restrict_warming_level` is used to limit the results to only datasets that are present in the csv used for calculating warming levels. You can also pass a dict to verify that a given warming level is reached.

Note that compared to `search`, the result of `search_data_catalog` is a dictionary with one entry per unique ID. A given unique ID might contain multiple datasets as per `intake-esm`'s definition, because it groups catalog lines per *id - domain - processing_level - xrfreq*. Thus, it would separate model data that exists at different frequencies.

Example 1: Multiple variables and frequencies + Historical and future

Let's start by searching for CMIP6 data that has subdaily precipitation, daily minimum temperature and the land fraction data. The main difference compared to searching for reference datasets is that in most cases, `match_hist_and_fut` will be required to match *historical* simulations to their future counterparts. This works for both CMIP5 and CMIP6 nomenclatures.

[ ]:
```
import xscen as xs

variables_and_freqs = {"tasmin": "D", "pr": "3H", "sftlf": "fx"}
other_search_criteria = {"institution": ["NOAA-GFDL"]}

cat_sim = xs.search_data_catalogs(
    data_catalogs=[f"{Path().absolute()}/samples/pangeo-cmip6.json"],
    variables_and_freqs=variables_and_freqs,
    other_search_criteria=other_search_criteria,
    match_hist_and_fut=True,
)

cat_sim
```

If required, at this stage, a dataset can be looked at in more details. If we examine the results (look at the 'date_start' and 'date_end' columns), we'll see that it successfully found historical simulations in the *CMIP* activity and renamed both their *activity* and *experiment* to match the future simulations.

[ ]:
```
cat_sim["ScenarioMIP_NOAA-GFDL_GFDL-CM4_ssp585_r1i1p1f1_gr1"].df
```

Example 2: Restricting results

The two previous search results were the same simulation, but on 2 different grids (`gr1` and `gr2`). If desired, `restrict_resolution` can be called to choose the finest or coarsest grid.

[ ]:
```
variables_and_freqs = {"tasmin": "D", "pr": "3H", "sftlf": "fx"}
other_search_criteria = {"institution": ["NOAA-GFDL"], "experiment": ["ssp585"]}

cat_sim = xs.search_data_catalogs(
    data_catalogs=[f"{Path().absolute()}/samples/pangeo-cmip6.json"],
    variables_and_freqs=variables_and_freqs,
    other_search_criteria=other_search_criteria,
    match_hist_and_fut=True,
    restrict_resolution="finest",
)

cat_sim
```

Similarly, if we search for historical NorESM2-MM data, we'll find that it has 3 members. If desired, `restrict_members` can be called to choose a maximum number of realization per model.

[ ]:
```
variables_and_freqs = {"tasmin": "D"}
other_search_criteria = {"source": ["NorESM2-MM"], "experiment": ["historical"]}

cat_sim = xs.search_data_catalogs(
    data_catalogs=[f"{Path().absolute()}/samples/pangeo-cmip6.json"],
    variables_and_freqs=variables_and_freqs,
    other_search_criteria=other_search_criteria,
    restrict_members={"ordered": 2},
)

cat_sim
```

Finally, `restrict_warming_level` can be used to be sure that the results either exist in `xscen`'s warming level database (if a boolean), or reach a given warming level.

[ ]:
```
variables_and_freqs = {"tasmin": "D"}

cat_sim = xs.search_data_catalogs(
    data_catalogs=[f"{Path().absolute()}/samples/pangeo-cmip6.json"],
    variables_and_freqs=variables_and_freqs,
    match_hist_and_fut=True,
    restrict_warming_level={
        "wl": 2
    },  # SSP126  gets eliminated, since it doesn't reach +2°C by 2100.
)

cat_sim
```

Example 3: Search for data that can be computed from what's available

`allow_resampling` and `allow_conversion` are powerful search tools to find data that doesn't explicitely exist in the catalog, but that can easily be computed.

[ ]:
```
cat_sim_adv = xs.search_data_catalogs(
    data_catalogs=[f"{Path().absolute()}/samples/pangeo-cmip6.json"],
    variables_and_freqs={"evspsblpot": "D", "tas": "YS"},
    other_search_criteria={"source": ["NorESM2-MM"], "processing_level": ["raw"]},
    match_hist_and_fut=True,
    allow_resampling=True,
    allow_conversion=True,
)
cat_sim_adv
```

If we examine the SSP5-8.5 results, we'll see that while it failed to find *evspsblpot*, it successfully understood that *tasmin* and *tasmax* can be used to compute it. It also understood that daily *tasmin* and *tasmax* is a valid search result for `{tas: YS}`, since it can be computed first, then aggregated to a yearly frequency.

[ ]:
```
cat_sim_adv["ScenarioMIP_NCC_NorESM2-MM_ssp585_r1i1p1f1_gn"].unique()
```

It's also possible to search for multiple frequencies at the same time by using a list of xrfreq.

[ ]:
```
cat_sim_adv_multifreq = xs.search_data_catalogs(
    data_catalogs=[f"{Path().absolute()}/samples/pangeo-cmip6.json"],
    variables_and_freqs={"tas": ["D", "MS", "YS"]},
    other_search_criteria={
        "source": ["NorESM2-MM"],
        "processing_level": ["raw"],
        "experiment": ["ssp585"],
    },
    match_hist_and_fut=True,
    allow_resampling=True,
    allow_conversion=True,
)
print(
    cat_sim_adv_multifreq[
        "ScenarioMIP_NCC_NorESM2-MM_ssp585_r1i1p1f1_gn"
    ]._requested_variable_freqs
)
```

Derived variables

The `allow_conversion` argument is built upon `xclim`'s virtual indicators module and `intake-esm`'s DerivedVariableRegistry in a way that should be seamless to the user. It works by using the methods defined in `xscen/xclim_modules/conversions.yml` to add a registry of *derived* variables that exist virtually through computation methods.

In the example above, we can see that the search failed to find *evspsblpot* within *NorESM2-MM*, but understood that *tasmin* and *tasmax* could be used to estimate it using `xclim`'s `potential_evapotranspiration`.

Most use cases should already be covered by the aforementioned file. The preferred way to add new methods is to submit a new indicator to xclim, and then to add a call to that indicator in `conversions.yml`. In the case where this

is not possible or where the transformation would be out of scope for `xclim`, the calculation can be implemented into `xscen/xclim_modules/conversions.py` instead.

Alternatively, if other functions or other parameters are required for a specific use case (e.g. using `relative_humidity` instead of `relative_humidity_from_dewpoint`, or using a different formula), then a custom YAML file can be used. This custom file can be referred to using the `conversion_yaml` argument of `search_data_catalogs`.

`.derivedcat` can be called on a catalog to obtain the list of DerivedVariable and the function associated to them. In addition, `._requested_variables` will display the list of variables that will be opened by the `to_dataset_dict()` function, including *DerivedVariables*.

WARNING

`_requested_variables` should NOT be modified under any circumstance, as it is likely to make `to_dataset_dict()` fail. To add some transparency on which variables have been **requested** and which are the **dependent** ones, `xscen` has added `_requested_variables_true` and `_dependent_variables`. This is very likely to be changed in the future.

[ ]:
```
cat_sim_adv["ScenarioMIP_NCC_NorESM2-MM_ssp585_r1i1p1f1_gn"].derivedcat
```

[ ]:
```
print(cat_sim_adv["ScenarioMIP_NCC_NorESM2-MM_ssp585_r1i1p1f1_gn"]._requested_variables)
print(
    f"Requested: {cat_sim_adv['ScenarioMIP_NCC_NorESM2-MM_ssp585_r1i1p1f1_gn']._
↪requested_variables_true}"
)
print(
    f"Dependent: {cat_sim_adv['ScenarioMIP_NCC_NorESM2-MM_ssp585_r1i1p1f1_gn']._
↪dependent_variables}"
)
```

INFO

`allow_conversion` currently fails if:

The requested DerivedVariable also requires a DerivedVariable itself.

The dependent variables exist at different frequencies (e.g. 'pr @1hr' & 'tas @3hr')

Creating a New Catalog from a Directory

Initialisation

The `create` argument of `ProjectCatalog` can be called to create an empty *ProjectCatalog* and a new set of JSON and CSV files.

By default, `xscen` will populate the JSON with generic information, defined in `catalog.esm_col_data`. That metadata can be changed using the `project` argument with entries compatible with the ESM Catalog Specification (refer to the link above). Usually, the most useful and common entries will be:

- title
- description

xscen will also instruct intake_esm to group catalog lines per *id - domain - processing_level - xrfreq*. This should be adequate for most uses. In the case that it is not, the following can be added to project:

- "aggregation_control": {"groupby_attrs": [list_of_columns]}

Other attributes and behaviours of the project definition can be modified in a similar way.

[ ]:
```
project = {
    "title": "tutorial-catalog",
    "description": "Catalog for the tutorial NetCDFs.",
}

PC = ProjectCatalog(
    str(output_folder / "tutorial-catalog.json"),
    create=True,
    project=project,
    overwrite=True,
)
```

[ ]:
```
# The metadata is stored in PC.esmcat
PC.esmcat
```

Appending new data to a ProjectCatalog

At this stage, the CSV is still empty. There are two main ways to populate a catalog with data:

- Using xs.ProjectCatalog.update_from_ds to append a Dataset and populate the catalog columns using metadata.
- Using xs.catutils.parse_directory to parse through existing NetCDF or Zarr data and decode their information based on file and directory names.

This tutorial will focus on catutils.parse_directory, as update_from_ds is moreso a function that will be called during a climate-scenario-generation workflow. See the *Getting Started* tutorial for more details on update_from_ds.

Parsing a directory

INFO

If you are an Ouranos employee, this section should be of limited use (unless you need to retroactively parse a directory containing exiting datasets). Please consult the existing Ouranos catalogs using xs.search_data_catalogs instead.

The `parse_directory <../xscen.rst#xscen.catutils.parse_directory>`__ function relies on analyzing patterns to adequately decode the filenames to store that information in the catalog.

- Patterns are a succession of column names in curly brackets. See below for examples. The pattern starts where the directory path stops.
- If necessary, read_from_file can be used to open the files and read metadata from global attributes. Refer to the API for Docstrings and usage.
- In cases where some column information is the same across all data, homogenous_info can be used to explicitely give an attribute to the datasets being processed.
- Anything that isn't filled will be marked as None.

The following example will search through the samples folder and infer information from the folder names. The filename is ignored, except its extension. The variable name and time bounds are read from the file itself.

[ ]:
```
from xscen.catutils import parse_directory

df = parse_directory(
    directories=[f"{Path().absolute()}/samples/tutorial/"],
    patterns=[
        "{activity}/{domain}/{institution}/{source}/{experiment}/{member}/{frequency}/{?:
↪_}.nc"
    ],
    homogenous_info={
        "mip_era": "CMIP6",
        "type": "simulation",
        "processing_level": "raw",
    },
    read_from_file=["variable", "date_start", "date_end"],
)
df
```

Unique Dataset ID

In addition to the parse itself, `parse_directory` will create a unique Dataset ID that can be used to easily determine one simulation from another. This can be edited with the `id_columns` argument of `parse_directory`, but by default, IDs are based on CMIP6's ID structure with additions related to regional models and bias adjustment:

- `{bias_adjust_project} _ {mip_era} _ {activity} _ {driving_model} _ {institution} _ {source} _ {experiment} _ {member} _ {domain}`

This utility can also be called by itself through `xs.catalog.generate_id()`.

INFO

When constructing IDs, empty columns will be skipped.

[ ]:
```
df.iloc[0]["id"]
```

Appending data using ProjectCatalog.update()

At this stage, `df` is a `pandas.DataFrame`. `ProjectCatalog.update` is used to append this data to the CSV file and save the results on disk.

[ ]:
```
PC.update(df)

PC
```

More on patterns and advanced features

The `patterns` argument acts as a reverse format string.

- The "_" format specifier (like in `{field:_}` allows matching a name containing underscores for this field. The path separators (/, \) are still excluded. Any format specifier supported by `parse are usable <https://github.com/r1chardj0n3s/parse>`__.
- Fields starting with a "?" will be ignored in the output. This allows to have readable patterns to identify parts we know exist, but do not want to be included in the metadata
- The `DATES` special field will match single dates or date bounds (see below).
- `{?:_}` is useful in filenames as a "wildcard" matching. For exammple: `{?:_}_{DATES}.nc` will read in the last "element" of the filename into `date_start` and `date_end`, ignoring all previous parts.

[ ]:
```
# Create fake files for the example:
root = Path(".").absolute() / "_data" / "parser_examples"
root.mkdir(exist_ok=True)

paths = [
    # Folder name includes underscore, single year implicitly means the full year
    "CCCma/CanESM2/day/tg_mean/tg_mean_1950.nc",
    # Fx frequency, no date bounds, strange model name
    "CCCma/CanESM-2/fx/sftlf/sftlf_fx.nc",
    # Bounds given as range at a monthly frequency
    "MIROC/MIROC6/mon/uas/uas_199901-200011.nc",
    # Version number included in the source name, range given a years
    "ERA/ERA5_v2/yr/heat_wave_frequency/hwf_2100-2399.nc",
]
for path in paths:
    (root / path).parent.mkdir(exist_ok=True, parents=True)
    with (root / path).open("w") as f:
        f.write("example")
```

Example 1 - wrong

The `variable` field does not allow underscores, so the first and last files are not parsed correctly.

Notice how the `DATES` field was parsed into `date_start` and `date_end`. It also matched with `fx`, returning `NaT` for both fields, as expected.

[ ]:
```
patt = "{institution}/{source}/{frequency}/{variable}/{?var}_{DATES}.nc"
parse_directory(directories=[root], patterns=[patt])
```

Example 2 - wrong again

We fixed the variable field by allowing underscores. We also modified the filename pattern to match any string, including underscores, except for the last element.

Notice how the "1950" part of `tg_mean` has been converted to `date_start='1950-01-01'` and `date_end='1950-12-31'`.

The `source` field does not allow underscores, so "ERA5_v2" is not parsed correctly. However, what we would want is rather to assign "v2" to the version field.

[ ]:
```
patt = "{institution}/{source}/{frequency}/{variable:_}/{?:_}_{DATES}.nc"
parse_directory(directories=[root], patterns=[patt])
```

Example 3 - Correct!

We added a second pattern that includes the `version` field.

[ ]:
```
patts = [
    "{institution}/{source}_{version}/{frequency}/{variable:_}/{?:_}_{DATES}.nc",
    "{institution}/{source}/{frequency}/{variable:_}/{?:_}_{DATES}.nc",
]
parse_directory(directories=[root], patterns=patts)
```

Example 4 - Filter on folder names

We can filter the results to include only some folders with the `dirglob` argument.

[ ]:
```
parse_directory(directories=[root], patterns=patts, dirglob="*/CanESM*")
```

Example 5 - Modifying metadata

We use the `cvs` (Controlled VocabularieS) argument here to replace some terms found in the paths by others we prefer.

Two replacement types are used : - simple : in the `source` column, all values of "CanESM-2" are replaced by "CanESM2" - complex : in the `institution` column, if the value "MIROC" is seen, it triggers the setting of "global" in this row's `domain` column, overriding whatever was already present in this field.

[ ]:
```
parse_directory(
    directories=[root],
    patterns=patts,
    cvs={
        "source": {"CanESM-2": "CanESM2"},
        "institution": {"MIROC": {"domain": "global"}},
    },
)
```

Example 6 : Even more complex field processing

In the preceding example, we used the `cvs` argument to replace values by others, or to trigger replacements based on values of other columns. The exact value must be matched and map to exact values. Another alternative to transform the parsed fields is to feed a function to the path parsing step. This is done by declaring a new "type" to the parser. In the following example, we'll implement a very useless transformation that reverses the letters of the institution.

[ ]:
```
from xscen.catutils import register_parse_type


@register_parse_type("rev")
def _reverse_word(text):
    return "".join(reversed(text))


patts_mod = [
    "{institution:rev}/{source}_{version}/{frequency}/{variable:_}/{?:_}_{DATES}.nc",
    "{institution:rev}/{source}/{frequency}/{variable:_}/{?:_}_{DATES}.nc",
]
parse_directory(directories=[root], patterns=patts_mod)
```

Restructuring catalogued files on disk

The opposite operation to `parse_directory` is also handled by `xscen.catutils`. In this section, we show how to create a Path from a xscen-extraced dataset or from a catalog entry.

Simple : template string and attributes

Given a dataset that was opened by `xs.extract_dataset` or `DataCatalog.to_dataset()`, we can easily construct a path from the xscen-added attributes.

[ ]:
```
# Open
ds = PC.search(variable="tas", experiment="ssp585").to_dataset()

path_template = "{institution}/{source}/{experiment}_{frequency}.nc"

print(path_template.format(**xs.utils.get_cat_attrs(ds)))
```

While this method is simple, it can't handle neither the list-like `variable` field nor the `date_start` and `date_end` datetime fields.

Complete : build_path

The `build_path <../xscen.rst#xscen.catutils.build_path>`__ function has a more complex interface to be used in more complex workflows.

The default parameters has a pretty good folder structure that depends on the columns `type` (usually one of simulation, reconstruction or station-obs) and `processing_level` (often raw, biasadjusted or something else).

[ ]:
```
xs.catutils.build_path(ds)
```

The folder schema can be passed explictly, as a dictionary with two entries: - “folders” : a list of fields to build the folder hierarchy. - “filename” : a list of fields to build the filename.

In both cases, a special “DATES” field can be given. It will be translated to the most efficient way to write the temporal bounds of the dataset.

[ ]:
```
custom_schema = {
    "folders": ["type", "institution", "source", "experiment"],
    "filename": ["variable", "DATES"],
}
xs.catutils.build_path(ds, schemas=custom_schema)
```

The function has more options:

- A “root” folder can be specified
- Other fields can be passed to override those in the data or fill for missing ones.

[ ]:
```
xs.catutils.build_path(ds, root=Path("/tmp"), domain="REG")
```

Above, we called the function with a dataset. In this case, the “facets” are extracted from various sources, with this priority (highest at the top):

1. Facets passed explicitly to `build_path` as keyword arguments
2. Attributes prefixed with “cat:”
3. Other Attributes
4. variable names, start and end date, and frequency, as extracted by `parse_from_date`.

But the function can also take a single dataframe row:

[ ]:
```
xs.catutils.build_path(PC.search(variable="tas", experiment="ssp585").df.iloc[0])
```

Or a full DataFrame/Catalog. In this case, the return value is a DataFrame, copy form the catalog, with a “new_path” column added.

[ ]:
```
# We show only three columns of the output catalog
xs.catutils.build_path(PC.search(variable="tas"))[["id", "path", "new_path"]]
```

This can be used in a workflow that renames or copies the files to their new name, usually using `shutil`.

[ ]:
```
import shutil as sh

# Create the destination folder
root = Path(".").absolute() / "_data" / "path_builder_examples"
root.mkdir(exist_ok=True)

# Get new names:
newdf = xs.catutils.build_path(PC, root=root)

# Copy files
for i, row in newdf.iterrows():
    Path(row["new_path"]).parent.mkdir(parents=True, exist_ok=True)
    sh.copy(row["path"], row["new_path"])
    print(f"Copied {row['path']}\n\tto {row['new_path']}")
```

(continues on next page)

(continued from previous page)

```
# Update catalog:
PC.df["path"] = newdf["new_path"]
PC.update()
```

2.4.2 Getting Started

This Notebook will go through all major steps of creating a climate scenario using `xscen`. These steps are:

- `search_data_catalogs` to find a subset of datasets that match a given project's requirements.
- `extract_dataset` to extract them.
- `regrid_dataset` to regrid all data to a common grid.
- `train` and `adjust` to bias correct the raw simulations.
- `compute_indicators` to compute a list of indicators.
- `climatological_op` and `spatial_mean` for spatio-temporal aggregation.
- `compute_deltas` to compute deltas.
- `ensemble_stats` for ensemble statistics.
- `clean_up` for minor adjustments that have to be made in preparation for the final product.

Initialisation

Typically, the first step should be to create a new *ProjectCatalog* to store the files that will be created during the process. More details on basic usage are provided in the Catalogs Notebook.

[ ]:
```
from pathlib import Path

import xscen as xs

# Folder where to put the data
output_folder = Path().absolute() / "_data"
output_folder.mkdir(exist_ok=True)

project = {
    "title": "example-gettingstarted",
    "description": "This is an example catalog for xscen's documentation.",
}

pcat = xs.ProjectCatalog(
    str(output_folder / "example-gettingstarted.json"),
    create=True,
    project=project,
    overwrite=True,
)
```

Searching a subset of datasets within *DataCatalogs*

INFO

At this stage, the search criteria should be for variables that will be **bias corrected**, not necessarily the variables required for the final product. For example, if `sfcWindfromdir` is the final product, then `uas` and `vas` should be searched for since these are the variables that will be bias corrected.

`xs.search_data_catalogs` is used to consult a list of *DataCatalogs* and find a subset of datasets that match given search parameters. More details on that function and possible usage are given in the *Understanding Catalogs* Notebook.

The function also plays the double role of preparing certain arguments for the extraction function, as detailed in the relevant *section of this tutorial*.

Due to how different reference datasets are from climate simulations, this function might have to be called multiple times and the results concatenated into a single dictionary.

For the purpose of this tutorial, temperatures and the land fraction from NorESM2-MM will be used:

[ ]:
```
variables_and_freqs = {"tas": "D", "sftlf": "fx"}
other_search_criteria = {
    "source": ["NorESM2-MM"],
    "processing_level": ["raw"],
    "experiment": "ssp245",
}

cat_sim = xs.search_data_catalogs(
    data_catalogs=[str(output_folder / "tutorial-catalog.json")],
    variables_and_freqs=variables_and_freqs,
    other_search_criteria=other_search_criteria,
    periods=[2001, 2002],
    match_hist_and_fut=True,
)

cat_sim
```

The result of `search_data_catalog` is a dictionary with one entry per unique ID. Note that a unique ID can be associated to multiple *intake datasets*, as is the case here, because `intake-esm` groups catalog lines per *id - domain - processing_level - xrfeq*.

[ ]:
```
cat_sim["CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp245_r1i1p1f1_example-region"].df
```

Extracting data

WARNING

It is heavily recommended to stop and analyse the results of `search_data_catalogs` before proceeding to the extraction function.

Defining the region

The region for a given project is defined using a dictionary with the relevant information to be used by `clisops.core.subset`. The required fields are as follows:

region =

```
"name": str,                    {this will overwrite the *domain* column in the catalog}
"method": str,                  [gridpoint, bbox, shape, sel]
"tile_buffer": float,    {approximate number of pixels used to expand the domain based␣
↪on model resolution}
**kwargs                        {other arguments to send `clisops`}
```

The argument *tile_buffer* (optional) is used to apply a buffer zone around the region that is adjusted dynamically according to model resolution during the extraction process (for *bbox* and *shape* only). This is useful to make sure that grid cells that only partially cover the region are selected too.

The documentation for the supported subsetting methods are in the following links:

- gridpoint
- bbox
- shape
- *sel* is simply a call to xarray

[ ]:
```
region = {
    "name": "example-region",
    "method": "bbox",
    "tile_buffer": 1.5,
    "lon_bnds": [-75, -74],
    "lat_bnds": [45, 46],
}
```

Preparing arguments for *xarray*

`xscen` makes use of `intake_esm`'s to_dataset_dict() for the extraction process, which will automatically compute missing variables as required. Also, this function manages Catalogs, IDs, and both NetCDF and Zarr files seamlessly. When the catalog is made of a single dataset, `to_dataset()` can be used instead to directly obtain an *xr.Dataset* instead of a dictionary.

There are a few key differences compared to using *xarray* directly, one of which being that it uses `xr.open_dataset`, even when multiple files are involved, with a subsequent call to `xr.combine_by_coords`. Kwargs are therefore separated in two:

- `xr_open_kwargs` is used for optional arguments in `xarray.open_dataset`.
- `xr_combine_kwargs` is used for optional arguments in `xarray.combine_by_coords`.

More information on possible kwargs can be obtained here: xarray.open_dataset & xarray.combine_by_coords

[ ]:
```
# Kwargs for xr.open_dataset
xr_open_kwargs = {"drop_variables": ["height", "time_bnds"], "engine": "h5netcdf"}

# Kwargs for xr.combine_by_coords
xr_combine_kwargs = {"data_vars": "minimal"}
```

Extraction function

Extraction is done on each dataset by calling `xs.extract_dataset()`. Since the output could have multiple frequencies, the function returns a python dictionary with keys following the output frequency.

- `catalog` is the *DataCatalog* to extract.
- `variables_and_freqs` is the same as previously used for `search_data_catalogs`.
- `periods` is used to extract specific time periods.
- `to_level` will change the *processing_level* of the output. Defaults to "extracted".
- `region`, `xr_open_kwargs`, and `xr_combine_kwargs` are described above.

NOTE: Calling the extraction function without passing by `search_data_catalogs` beforehand is possible, but will not support *DerivedVariables*.

NOTE

`extract_dataset` currently only accepts a single unique ID at a time.

[ ]:
```
# Example with a single simulation
ds_dict = xs.extract_dataset(
    catalog=cat_sim["CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp245_r1i1p1f1_example-region"],
    variables_and_freqs=variables_and_freqs,
    periods=[2001, 2002],
    region=region,
    xr_open_kwargs=xr_open_kwargs,
    xr_combine_kwargs=xr_combine_kwargs,
)

ds_dict
```

Saving files to disk

`extract_dataset` does not actually *save* anything to disk. It simply opens and prepares the files as per requested, using lazy computing. The result is a python dictionary containing the results, separated per *xrfreq*.

`xscen` has two functions for the purpose of saving data: `save_to_netcdf` and `save_to_zarr`. If possible for a given project, it is strongly recommended to use Zarr files since these are often orders of magnitude faster to read and create compared to NetCDF. They do have a few quirks, however:

- Chunk size must separate the dataset in *exactly* equal chunks (with the exception of the last). While it is recommended to calculate ideal chunking and provide them explicitely to the function, `io.estimate_chunks()` can be used to automatically estimate a decent chunking. In a similar way, `io.subset_maxsize()` can be used to roughly cut a dataset along the *time* dimension into subsets of a given size (in Gb), which is especially useful for saving NetCDF files. Chunk sizes can be passed to the two saving functions in a dictionary. Spatial dimensions can be generalized as `'X'` and `'Y'`, which will be mapped to the *xarray.Dataset*'s actual grid type's dimension names.
- Default behaviour for a Zarr is to act like a directory, with each new variable being assigned a subfolder. This is great when all required variables have the same dimensions and frequency, but will crash otherwise. If you have daily *tasmax* and subdaily *pr*, for example, they need to be assigned different paths.

Updating the catalog

`intake-esm` will automatically copy the catalog's entry in the dataset's metadata, with a `cat:attr` format. Where appropriate, `xscen` updates that information to keep the metadata up to date with the manipulations. `ProjectCatalog.update_from_ds` will in turn read the metadata of a Dataset and fill in the information into a new catalog entry.

This loop means that upon completing a step in the creation of a climate scenario, `ProjectCatalog.update_from_ds()` can be called to update the catalog.

[ ]:
```
for ds in ds_dict.values():
    filename = str(
        output_folder
        / f"{ds.attrs['cat:id']}.{ds.attrs['cat:domain']}.{ds.attrs['cat:processing_level
↪']}.{ds.attrs['cat:frequency']}.zarr"
    )
    chunks = xs.io.estimate_chunks(ds, dims=["time"], target_mb=50)
    xs.save_to_zarr(ds, filename, rechunk=chunks, mode="o")

    # Strongly suggested to update the project catalog AFTER you save to disk, in case␣
↪it crashes during the process
    pcat.update_from_ds(ds=ds, path=filename, info_dict={"format": "zarr"})

pcat.df
```

Simplifying the call to extract_dataset() with search_data_catalogs()

When a catalog was produced using `search_data_catalogs`, `xscen` will automatically save the requested periods and frequencies, in addition to *DerivedVariables*. This means that these items do not need to be included during the call to `extract_dataset` and make it possible to extract datasets that have different requirements (such as reference datasets and future simulations).

[ ]:
```
cat_sim[
    "CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp245_r1i1p1f1_example-region"
]._requested_periods
```

[ ]:
```
print(
    cat_sim[
        "CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp245_r1i1p1f1_example-region"
    ]._requested_variables_true
)
print(
    cat_sim[
        "CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp245_r1i1p1f1_example-region"
    ]._requested_variable_freqs
)
```

Since `cat_sim` contains multiple datasets, extracting the data should be done by looping on `.items()` or `.values()`. Also, since 'CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp126_r1i1p1f1_example-region' was extracted in the previous step, `pcat.exists_in_cat` can be used to skip re-extracting.

[ ]:
```
for key, dc in cat_sim.items():
    if not pcat.exists_in_cat(id=key, processing_level="extracted"):
```

(continues on next page)

(continued from previous page)

```
        dset_dict = xs.extract_dataset(
            catalog=dc,
            region=region,
            xr_open_kwargs=xr_open_kwargs,
            xr_combine_kwargs=xr_combine_kwargs,
        )

        for ds in dset_dict.values():
            filename = str(
                output_folder
                / f"{ds.attrs['cat:id']}.{ds.attrs['cat:domain']}.{ds.attrs['cat:
↪processing_level']}.{ds.attrs['cat:frequency']}.zarr"
            )
            chunks = xs.io.estimate_chunks(ds, dims=["time"], target_mb=50)
            xs.save_to_zarr(ds, filename, rechunk=chunks, mode="o")

            # Strongly suggested to update the project catalog AFTER you save to disk,␣
↪in case it crashes during the process
            pcat.update_from_ds(ds=ds, path=filename, info_dict={"format": "zarr"})
```

Regridding data

NOTE

Regridding in `xscen` is built upon `xESMF`. For more information on basic usage and available regridding methods, please consult their documentation. Their masking and extrapolation tutorial is of particular interest.

More details on the regridding functions themselves can be found within the ESMPy and ESMF documentation.

The only requirement for using datasets in `xESMF` is that they contain *lon* and *lat*, with *mask* as an optional data variable. Using these, the package can manage both regular and rotated grids. The main advantage of `xESMF` compared to other tools such as *scipy*'s *griddata*, in addition to the fact that the methods are climate science-based, is that the transformation weights are calculated once and broadcasted on the *time* dimension.

Preparing the destination grid

`xscen` currently does not explicitely support any function to create a destination grid. If required, however, `xESMF` itself has utilities that can easily create custom regular grids, such as `xesmf.util.cf_grid_2d`.

[ ]:
```
import xesmf

ds_grid = xesmf.util.cf_grid_2d(-75, -74, 0.25, 45, 48, 0.55)

# cf_grid_2d does not set the 'axis' attribute
ds_grid["lon"].attrs["axis"] = "X"
ds_grid["lat"].attrs["axis"] = "Y"

# xscen will use the domain to re-assign attributes, so it is important to set it up for␣
↪custom grids like this
```

(continues on next page)

(continued from previous page)

```
ds_grid.attrs["cat:domain"] = "finer-grid"
ds_grid
```

Masking grid cells

Masks can be used on both the original grid and the destination grid to ignore certain grid cells during the regridding process. These masks follow the ESMF convention, meaning that the mask is a variable within the Dataset, named *mask* and comprised of 0 and 1.

`xs.create_mask` will create an adequate DataArray, following the instructions given by *mask_args*. In the case of variables that have a time component, the first timestep will be chosen.

```
mask_args:
   - 'variable'         (optional)
   - 'where_operator'   (optional)
   - 'where_threshold'  (optional)
   - 'mask_nans': bool
```

[ ]:
```
# Will mask all pixels that do not match these requirements (at least 25% land)
mask_args = {
    "variable": "sftlf",
    "where_operator": ">",
    "where_threshold": 25,
    "mask_nans": True,
}

# to_dataset() will open the dataset, as long as the search() gave a single result.
ds_example = pcat.search(
    id="CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp245_r1i1p1f1_example-region",
    processing_level="extracted",
    variable="sftlf",
).to_dataset()

# Masking function
ds_example["mask"] = xs.create_mask(ds_example, mask_args=mask_args)
```

[ ]:
```
import matplotlib.patches as patches
import matplotlib.pyplot as plt

# Plot sftlf
ax = plt.subplot(121)
ds_example.sftlf.plot.imshow()
plt.title("sftlf")

# Plot the mask
plt.subplot(122)
ds_example.mask.plot.imshow()
plt.title("mask")
```

Preparing arguments for xESMF.Regridder

NOTE

xESMF's API appears to be broken on their ReadTheDocs. For a list of available arguments and options in `Regridder()`, please consult their Github page directly.

`xESMF.Regridder` is the main utility that computes the transformation weights and performs the regridding. It is supported by many optional arguments and methods, which can be called in `xscen` through `regridder_kwargs`.

Available options are:

```
 method: str
     Regridding method. More details are given within the ESMF documentation and xESMF␣
↪tutorials.
    - 'bilinear'  (Default)
    - 'nearest_s2d'
    - 'nearest_d2s'
    - 'conservative'
    - 'conservative_normed'
    - 'patch'

extrap_method: str
    Extrapolation method. Defaults to None.
    - 'inverse_dist'
    - 'nearest_s2d'

extrap_dist_exponent: float
    Exponent to raise the distance to when calculating weights for extrapolation.
    Defaults to 2.0.

extrap_num_src_pnts : int, optional
    The number of source points to use for the extrapolation methods that use more than␣
↪one source point.
    Defaults to 8

unmapped_to_nan: boolean, optional
    Set values of unmapped points to `np.nan` instead of the ESMF default of 0.
    Defaults to True.

periodic: boolean, optional
    Only really useful for global grids with non-conservative regridding.
```

Other options exist in `ESMF/ESMPy`, but not `xESMF`. As they get implemented, they should automatically get supported by `xscen`.

NOTE

Some utilities that exist in `xESMF` have not yet been explicitely added to `xscen`. If *conservative* regridding is desired, for instance, some additional scripts might be required on the User's side to create the lon/lat boundaries

[ ]:
```
regridder_kwargs = {"extrap_method": "inverse_dist"}
```

Regridding function

Regridding for a Dataset is done through `xs.regrid_dataset`, which manages calls to `xESMF.Regridder` and makes sure that the output is CF-compliant.

- `weights_location` provides a path where to save the regridding weights (NetCDF file). This file (alongside `reuse_weights=True`) is used by `xESMF` to reuse the transformation weights between datasets that are deemed to have the same grid and vastly improve the speed of the function.
- `intermediate_grids` can be called to perform the regridding process in multiple steps. This is recommended when the jump in resolution is very high between the original and destination grid (e.g. from 3° to 0.08°).
- `ds_grid` & `regridder_kwargs` are described above.

[ ]:
```
# to_dataset_dict() is called to cast the search results as xr.Dataset objects
# frequency="^(?!fx$).*$" is used to exclude fixed fields from the results
ds_dict = pcat.search(
    processing_level="extracted", frequency="^(?!fx$).*$", domain="example-region"
).to_dataset_dict()

mask_args = {
    "variable": "sftlf",
    "where_operator": ">",
    "where_threshold": 25,
    "mask_nans": True,
}

for ds in ds_dict.values():
    # Add a mask on the original grid.
    ds["mask"] = xs.create_mask(
        pcat.search(
            id=ds.attrs["cat:id"], processing_level="extracted", variable="sftlf"
        ).to_dataset(),
        mask_args=mask_args,
    )

    # Regridding function
    ds_regrid = xs.regrid_dataset(
        ds=ds,
        weights_location=str(output_folder / "gs-weights"),
        ds_grid=ds_grid,
        regridder_kwargs=regridder_kwargs,
    )

    # Save to zarr
    filename = str(
        output_folder
        / f"{ds_regrid.attrs['cat:id']}.{ds_regrid.attrs['cat:domain']}.{ds_regrid.attrs[
↪'cat:processing_level']}.{ds_regrid.attrs['cat:frequency']}.zarr"
    )
    chunks = xs.io.estimate_chunks(ds, dims=["time"], target_mb=50)
```

(continues on next page)

(continued from previous page)

```
    xs.save_to_zarr(ds=ds_regrid, filename=filename, rechunk=chunks, mode="o")
    pcat.update_from_ds(ds_regrid, path=filename, format="zarr")
```

[ ]:
```
import matplotlib.patches as patches

plt.figure(figsize=[15, 5])

vmin = float(ds.tas.isel(time=0).min())
vmax = float(ds.tas.isel(time=0).max())

ax = plt.subplot(131)
ds.tas.isel(time=0).plot.imshow(vmin=vmin, vmax=vmax)
plt.title("tas: original grid")
rect = patches.Rectangle(
    (-75, 45), 1, 2.75, linewidth=1, edgecolor="r", facecolor="none"
)
ax.add_patch(rect)

ax = plt.subplot(132)
(ds.tas.isel(time=0).where(ds.mask == 1)).plot.imshow(vmin=vmin, vmax=vmax)
rect = patches.Rectangle(
    (-75, 45), 1, 2.75, linewidth=1, edgecolor="r", facecolor="none"
)
ax.add_patch(rect)
plt.title("tas: original + mask")
plt.tight_layout()

plt.subplot(133)
ds_regrid.tas.isel(time=0).plot.imshow(vmin=vmin, vmax=vmax)
plt.title("tas: regridded with mask + extrapolation")
plt.tight_layout()
```

Bias adjusting data

NOTE

Bias adjustment in `xscen` is built upon `xclim.sdba`. For more information on basic usage and available methods, please consult their documentation.

Preparing arguments for xclim.sdba

Many optional arguments are used by `xclim.sdba` during the training and adjustment processes. These options heavily depend on the bias adjustment method used, so it is recommended to consult their documentation before proceeeding further.

These arguments can be sent by using `xclim_train_kwargs` and `xclim_adjust_kwargs` during the call to `xs.train` and `xs.adjust` respectively.

```
[ ]: xclim_train_args = {"kind": "+", "nquantiles": 50}

xclim_adjust_args = {"detrend": 3, "interp": "linear", "extrapolation": "constant"}
```

Bias adjustment function

Bias adjustment is done through `xs.train` and `xs.adjust`. They are kept separate to account for cases where a voluminous dataset would require saving after the training step.

The arguments to *train()* are:

- `dref` and `dhist` indicate the reference and historical datasets.
- `var` indicates which variable to bias correct.
- `period` defines the period used for building the transfer function.
- `method` indicates which bias adjusting method to call within `xclim.sdba`.
- `maximal_calendar` instructs on which calendar to use, following this hierarchy: 360_day < noleap < standard < all_leap
- `adapt_freq` is used for bias adjusting the frequency of dry/wet days (precipitation only).
- `jitter_under` adds a random noise under a given threshold.
- `jitter_over`adds a random noise over a given threshold.
- `xclim_train_kwargs` is described above.

The arguments to *adjust()* are:

- `dtrain` is the result of `biasadjust.train`.
- `dsim` is the simulation to bias adjust.
- `periods` defines the period(s) to bias adjust.
- `xclim_adjust_kwargs` is described above.

NOTE

These functions currently do not support multiple variables due to the fact that train and adjust arguments might vary. The function must be called separately for every variable.

```
[ ]: ds_dict = pcat.search(processing_level="regridded").to_dataset_dict()

# # Open the reference dataset, in this case ERA5-Land
ds_ref = pcat.search(processing_level="extracted", source="ERA5-Land").to_dataset()

# Currently, only a single variable can be bias adjusted at a time
variables = ["tas"]
for v in variables:
    for ds in ds_dict.values():
        # Train
        ds_train = xs.train(
            dref=ds_ref,
            dhist=ds,
```

(continues on next page)

(continued from previous page)

```
            var=["tas"],
            period=["1981", "2010"],
            xclim_train_args=xclim_train_args,
        )

        # Adjust
        ds_adj = xs.adjust(
            dtrain=ds_train,
            dsim=ds,
            periods=["1981", "2050"],
            bias_adjust_institution="Ouranos",  # add new attribute cat:bias_adjust_
↪institution
            bias_adjust_project="xscen-tutorial",  # add new attribute cat:bias_adjust_
↪project
            xclim_adjust_args=xclim_adjust_args,
        )

        # Save to zarr
        filename = str(
            output_folder
            / f"{ds_adj.attrs['cat:id']}.{ds_adj.attrs['cat:domain']}.{ds_adj.attrs['cat:
↪processing_level']}.{ds_adj.attrs['cat:frequency']}.zarr"
        )
        chunks = xs.io.estimate_chunks(ds_adj, dims=["time"], target_mb=50)
        xs.save_to_zarr(ds=ds_adj, filename=filename, rechunk=chunks, mode="o")
        pcat.update_from_ds(ds_adj, path=filename, format="zarr")
```

[ ]:
```
ds = pcat.search(
    processing_level="regridded",
    variable="tas",
    id="CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp245_r1i1p1f1_example-region",
).to_dataset()
ds_adj = pcat.search(
    processing_level="biasadjusted",
    variable="tas",
    id="CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp245_r1i1p1f1_example-region",
).to_dataset()

vmin = min(
    [
        float(ds_ref.tas.sel(time=slice("1981", "2010")).mean(dim="time").min()),
        float(ds.tas.sel(time=slice("1981", "2010")).mean(dim="time").min()),
    ]
)
vmax = max(
    [
        float(ds_ref.tas.sel(time=slice("1981", "2010")).mean(dim="time").max()),
        float(ds.tas.sel(time=slice("1981", "2010")).mean(dim="time").max()),
    ]
)

fig = plt.figure(figsize=(15, 5))
```

(continues on next page)

(continued from previous page)

```
plt.subplot(141)
ds_ref.tas.sel(time=slice("1981", "2010")).mean(dim="time").transpose(
    "lat", ...
).plot.imshow(vmin=vmin, vmax=vmax)
plt.title("tas - Ref")

plt.subplot(142)
ds.tas.sel(time=slice("1981", "2010")).mean(dim="time").transpose(
    "lat", ...
).plot.imshow(vmin=vmin, vmax=vmax)
plt.title("tas - Raw")

plt.subplot(143)
ds_adj.tas.sel(time=slice("1981", "2010")).mean(dim="time").transpose(
    "lat", ...
).plot.imshow(vmin=vmin, vmax=vmax)
plt.title("tas - Bias adjusted")
plt.tight_layout()

plt.subplot(144)
(
    ds_adj.tas.sel(time=slice("1981", "2010")).mean(dim="time")
    - ds_ref.tas.sel(time=slice("1981", "2010")).mean(dim="time")
).transpose("lat", ...).plot.imshow(vmin=-1, vmax=1, cmap="RdBu_r")
plt.title("Bias (°C)")
plt.tight_layout()
```

Computing indicators

NOTE

`xscen` relies heavily on `xclim`'s YAML support for calculating indicators. For more information on how to build the YAML file, consult this Notebook.

`xs.compute_indicators` makes use of *xclim*'s indicator modules functionalities to compute a given list of indicators. It is called by either using:

- The path to a YAML file structured in a way compatible with *xclim*'s `build_indicator_module_from_yaml`
- An indicator module directly
- A sequence of indicators
- A sequence of tuples as returned by calling `iter_indicators()` on an indicator module.

Same as the extraction function, since the output could have multiple frequencies, the function returns a python dictionary with the output frequency as keys. The inputs of `xs.compute_indicators` are:

- `ds` is the *xr.Dataset* containing the required variables.
- `indicators` instructs on which indicator(s) to compute. It can be a number of things, as listed above.
- `periods` is a list of [start, end] of continuous periods to be considered.

This example will use a simple YAML file structured like this:

```
realm: atmos
indicators:
  growing_degree_days:
    base: growing_degree_days
  tg_min:
    base: tg_min
```

[ ]:
```
ds_dict = pcat.search(processing_level="biasadjusted").to_dataset_dict()

for ds in ds_dict.values():
    # Output is dict, but it has only one frequency.
    _, ds_ind = xs.compute_indicators(
        ds=ds,
        indicators=Path().absolute() / "samples" / "indicators.yml",
    ).popitem()

    # Save the results
    filename = str(
        output_folder
        / f"{ds_ind.attrs['cat:id']}.{ds_ind.attrs['cat:domain']}.{ds_ind.attrs['cat:
↪processing_level']}.{ds_ind.attrs['cat:frequency']}.zarr"
    )
    chunks = xs.io.estimate_chunks(ds, dims=["time"], target_mb=50)
    xs.save_to_zarr(ds_ind, filename, rechunk=chunks, mode="o")

    # Strongly suggested to update the project catalog AFTER you save to disk, in case␣
↪it crashes during the process
    pcat.update_from_ds(ds=ds_ind, path=filename, format="zarr")
```

[ ]:
```
display(ds_ind)
```

Spatio-temporal aggregation

Climatological operations

`xs.climatological_op` is used to perform *n*-year operations over `ds.time.dt.year`.

NOTE: The aggregation is over *year*, **not over time**. For example, if given monthly data, the climatological operation will be computed separately for January, February, etc. This means that the data should already be aggregated at the required frequency, for example using `xs.compute_indicators` to compute yearly, seasonal, or monthly indicators.

The function call requires a `xr.Dataset` and argument `op` specifies the operation to perform. It can be any of the following: `['max', 'mean', 'median', 'min', 'std', 'sum', 'var', 'linregress']`.

The optional arguments are as follows:

- `window` indicates how many year to use for the average. Uses all available years by default.
- `min_period` minimum number of years required for a value to be computed durring the `rolling` operation.
- `stride` indicates the stride (in years) at which to provide an output.
- `periods` is a list of [start, end] of continuous periods to be considered.

Additional arguments allow to control the output of the function by automatically renaming variables to reflect the operation performed, restructuring the output dataset and setting the `to_level` attribute.

In the following example, we will use `op='mean'`.

[ ]:
```
ds_dict = pcat.search(processing_level="indicators").to_dataset_dict()

for key, ds in ds_dict.items():
    ds_mean = xs.climatological_op(
        ds=ds,
        op="mean",
        window=30,
        stride=10,
        rename_variables=False,
        to_level="30yr-climatology",
        horizons_as_dim=False,
    )

    # Save to zarr
    filename = str(
        output_folder
        / f"{ds_mean.attrs['cat:id']}.{ds_mean.attrs['cat:domain']}.{ds_mean.attrs['cat:
↪processing_level']}.{ds_mean.attrs['cat:frequency']}.zarr"
    )
    xs.save_to_zarr(ds=ds_mean, filename=filename, mode="o")
    pcat.update_from_ds(ds_mean, path=filename, format="zarr")
```

[ ]:
```
display(ds_mean)
```

Horizon coordinate and time dimension

Even if no `stride` is called, `xs.climatological_op` will substantially change the nature of the `time` dimension, because it now represents an aggregation over time. While no standards exist on how to reflect that in a dataset, the following was chosen for `xscen`:

- `time` corresponds to the first timestep of each temporal average.
- `horizon` is a new coordinate that either follows the format YYYY-YYYY or a warming-level specific nomenclature.
- The `cat:frequency` and `cat:xrfreq` attributes remain unchanged.

Alternatively, setting the `horizons_as_dim` argument to *True* will rearrange the dataset with a new dimension `horizon` and a dimension named according to the temporal aggregation when it is `month` or `season`, but omitting the singleton dimension `year`. The time stamps are conserved in the `time` coordinate as an array with those new dimensions.

[ ]:
```
print(f"time: {ds_mean.time.values}")
print(f"horizon: {ds_mean.horizon.values}")
print(f"cat:xrfreq attribute: {ds_mean.attrs['cat:xrfreq']}")
```

Computing deltas

`xs.compute_deltas` is pretty self-explanatory. However, note that this function relies on the `horizon` coordinate described above and, thus, is intended to be performed following some kind of temporal aggregation.

It has the following arguments:

- `reference_horizon` indicates which horizon to use as reference.
- `kind` is either "+", "/", or "%" for absolute, relative, or percentage deltas respectively. This argument can also be a dictionary, with the keys corresponding to data variables.

```
[ ]: ds_dict = pcat.search(processing_level="30yr-climatology").to_dataset_dict()

for key, ds in ds_dict.items():
    ds_delta = xs.compute_deltas(
        ds=ds,
        reference_horizon="1981-2010",
        kind={"growing_degree_days": "%", "tg_min": "+"},
        to_level="deltas",
    )

    # Save to zarr
    filename = str(
        output_folder
        / f"{ds_delta.attrs['cat:id']}.{ds_delta.attrs['cat:domain']}.{ds_delta.attrs[
↪'cat:processing_level']}.{ds_delta.attrs['cat:frequency']}.zarr"
    )
    chunks = xs.io.estimate_chunks(ds, dims=["time"], target_mb=50)
    xs.save_to_zarr(ds=ds_delta, filename=filename, rechunk=chunks, mode="o")
    pcat.update_from_ds(ds_delta, path=filename, format="zarr")
```

```
[ ]: print(f"Deltas over {ds_delta.horizon.values}")
display(ds_delta.tg_min_delta_1981_2010.isel(lon=0, lat=0).values)
```

Spatial mean

`xs.spatial_mean` is used to compute the spatial average over a given region, using various methods. The argument `call_clisops` can also be used to subset the domain prior to the averaging.

- `method: cos-lat` will perform an average operation over the spatial dimensions, accounting for changes in grid cell area along the 'lat' coordinate.
- `method: interp_centroid` will perform an interpolation towards given coordinates or towards the centroid of a region.
  - `kwargs` is used to sent arguments to `.interp()`, including `lon` and `lat`.
  - `region` can alternatively be used to send a gridpoint, bbox, or shapefile and compute the centroid. This argument is a dictionary that follows the same requirements as the one for `xs.extract` described previously.
- `method: xesmf` will perform a call to *xESMF*'s SpatialAverager. This method is the most precise, especially for irregular regions, but can be much slower.
  - `kwargs` is used to sent arguments to `xesmf.SpatialAverager`.

- `region` is used to send a bbox or shapefile to the `SpatialAverager`. This argument is a dictionary that follows the same requirements as the one for `xs.extract` described previously.
- `simplify_tolerance` is a float that can be used to change the precision (in degree) of a shapefile before sending it to `SpatialAverager`.

[ ]:
```
ds_dict = pcat.search(processing_level="deltas", domain="finer-grid").to_dataset_dict()

for key, ds in ds_dict.items():
    ds_savg = xs.spatial_mean(
        ds=ds,
        method="interp_centroid",
        kwargs={"method": "linear", "lon": -74.5, "lat": 47},
        to_domain="aggregated",
    )

    # Save to zarr
    filename = str(
        output_folder
        / f"{ds_savg.attrs['cat:id']}.{ds_savg.attrs['cat:domain']}.{ds_savg.attrs['cat:
↪processing_level']}.{ds_savg.attrs['cat:frequency']}.zarr"
    )
    chunks = xs.io.estimate_chunks(ds, dims=["time"], target_mb=50)
    xs.save_to_zarr(ds=ds_savg, filename=filename, rechunk=chunks, mode="o")
    pcat.update_from_ds(ds_savg, path=filename, format="zarr")
```

[ ]:
```
# Aggregated deltas over the study area
display(ds_savg)
```

Ensemble statistics

Weights

Typically, if an ensemble is inhomogeneous (uneven number of realizations per model, mix of GCMs and RCMs, etc.), the first step should be to call `xs.generate_weights` to create an adequate guess of what the weights should be between the various datasets. Do note, however, that this function does not replace an explicit assessment of the performance or independence of the simulations, and the results provided should be taken with a grain of salt.

The arguments are as follows:

- `independence_level` instruct on which weighting scheme to use and strongly influences the outputs. One of 'model', 'GCM', 'institution'.
- `experiment_weights` can be used to assign a given total weight to each experiment (currently only supports giving 1 to each experiment).
- `skipna` instructs on whether the weights should account for simulations with missing data.
- `v_for_skipna` is the variable to use in the case of `skipna=False`.
- `standardize` to make the weights sum to 1 for each instance of 'horizon' or 'time'.

NOTE

`generate_weights` was built with `xscen` in mind, and thus relies on the `cat:` attributes automatically generated by `intake-esm` when data is loaded from a catalog. In the case of data generated elsewhere, the required and recommended attributes should minimally be added to the dataset before using this function.

[ ]:
```
# We don't have many simulations in this example to perform ensemble statistics, but we
↪'ll use the two SSP2-4.5 realizations
datasets = pcat.search(
    processing_level="deltas", domain="aggregated", experiment="ssp245"
).to_dataset_dict()

weights = xs.generate_weights(datasets, independence_level="model")
display(weights)
```

Ensemble stats

`xs.ensemble_stats` creates an ensemble out of many datasets and computes statistics on that ensemble (min, max, mean, percentiles, etc.) using the `xclim.ensembles` module. The inputs can be given in the form of a list or a dictionary of xr.Dataset or of paths.

The arguments are as follows:

- `statistics` is a dictionary that instructs on which `xclim.ensembles` statistics to call. It follows the format {function: arguments}.
- `weights` is used to weight the various datasets, if needed.
- `common_attrs_only`: `xclim.ensembles.create_ensemble` copies the attributes from the first dataset, but this might not be representative of the new ensemble. If `common_attrs_only` is True, it only keeps the global attributes that are the same for all datasets and generates a new ID.
- `create_kwargs`: If given a set of paths, `xclim.ensembles.create_ensemble` will ignore the chunking on disk and open the datasets with only a chunking along the new `realization` dimension. Thus, for large datasets, this should be used to explicitely specify chunks.

[ ]:
```
ens_stats = xs.ensemble_stats(
    datasets=datasets,
    statistics={
        "ensemble_percentiles": {"split": False}
    },  # should be an existing function in xclim.ensembles
    weights=weights,
    common_attrs_only=True,
)

path = output_folder / f"ensemble_{ens_stats.attrs['cat:id']}.zarr"
xs.save_to_zarr(ens_stats, filename=path, mode="o")
pcat.update_from_ds(ds=ens_stats, path=path, format="zarr")
```

[ ]:
```
display(ens_stats)
```

Clean up

At any time, such as after bias adjustment, `xs.clean_up` can be called to perform a number of small modifications to the datasets. That function can:

- convert the variables to non-CF units using `xs.utils.change_units`
- call the `xs.utils.maybe_unstack` function
- convert the calendar and interpolate over missing dates
- remove, remove everything but, and/or add a list of attributes
- change the prefix of the catalog attrs (by default: `cat:`)

in that order.

Calendars

During the bias adjustment step, it is frequent to convert the calendar to 'noleap'. However, once that step has been processed, we might want to put back all the February 29th (or other missing data in the case of '360_day' calendar). This can be done using the `convert_calendar_kwargs` argument of `xs.clean_up`, which passes a dictionary to `xclim.core.calendar.convert_calendar`.

Usually, we want to linearly interpolate the missing temperatures, but put 0 mm/day for missing precipitation. If our dataset has many variables, the `missing` argument (for `convert_calendar`) can be set for each variable with `missing_by_var`. If `missing_by_var` is given 'interpolate', the missing data will be filled with NaNs, then linearly interpolated over time.

Eg. `{'tasmax':'interpolate', 'pr':[0]}`

[ ]:
```
convert_calendar_kwargs = {"target": "standard"}
missing_by_var = {"tas": "interpolate"}
```

Attributes

We might want to add, remove or modify the attributes.

It is possible to write a list of attributes to remove with `attrs_to_remove`, or a list of attributes to keep and remove everything else with `remove_all_attrs_except`. Both take the shape of a dictionnary where the keys are the variables (and 'global' for global attrs) and the values are the list.

The element of the list can be exact matches for the attribute names or use the same regex matching rules as `intake_esm`:

- ending with a '*' means checks if the substring is contained in the string
- starting with a '^' means check if the string starts with the substring.

Attributes can also be added to datasets using `add_attrs`. This is a dictionary where the keys are the variables and the values are a another dictionary of attributes.

It is also possible to modify the catalogue prefix 'cat:' by a new string with `change_attr_prefix`. Don't use this if this is not the last step of your workflow.

[ ]:
```
attrs_to_remove = {
    "tas": ["name*"]
}  # remove tas attrs that contain the substring 'name'
```

(continues on next page)

(continued from previous page)

```
remove_all_attrs_except = {
    "global": ["^cat:"]
}  # remove all the global attrs EXCEPT for the one starting with cat:
add_attrs = {
    "tas": {"notes": "some crucial information"}
}  # add a new tas attribute named 'notes' with value 'some crucial information'
change_attr_prefix = "dataset:"  # change /cat to dataset:
```

[ ]:
```
ds = pcat.search(
    processing_level="biasadjusted", variable="tas", experiment="ssp245", member="r1.*"
).to_dataset()

ds_clean = xs.clean_up(
    ds=ds,
    variables_and_units={"tas": "degC"},  # convert units
    convert_calendar_kwargs=convert_calendar_kwargs,
    missing_by_var=missing_by_var,
    attrs_to_remove=attrs_to_remove,
    remove_all_attrs_except=remove_all_attrs_except,
    add_attrs=add_attrs,
    change_attr_prefix=change_attr_prefix,
)
```

[ ]:
```
from xclim.core.calendar import get_calendar

# Inspect calendars and the interpolated values
print("Initial calendar: ", get_calendar(ds.time))
print(ds.time.sel(time=slice("2000-02-28", "2000-03-01")).values)
print(ds.tas.sel(time=slice("2000-02-28", "2000-03-01")).isel(lat=1, lon=1).values)

print("Final calendar: ", get_calendar(ds_clean.time))
print(ds_clean.time.sel(time=slice("2000-02-28", "2000-03-01")).values)
print(
    ds_clean.tas.sel(time=slice("2000-02-28", "2000-03-01")).isel(lat=1, lon=1).values
)
print("")
print("Inspect initial attributes")
display(ds)

print("")
print("Inspect final attributes")
display(ds_clean)
```

[ ]:
```
from pathlib import Path

import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

import xscen as xs
```

(continues on next page)

(continued from previous page)

```
output_folder = Path().absolute() / "_data"

# Create a project Catalog
project = {
    "title": "example-diagnostics",
    "description": "This is an example catalog for xscen's documentation.",
}

pcat = xs.ProjectCatalog(
    str(output_folder / "example-diagnostics.json"),
    create=True,
    project=project,
    overwrite=True,
)
```

2.4.3 Diagnostics

It can be useful to perform a various diagnostic tests in order to check that the data that was produced is as expected. Diagnostics can also help us assess bias adjustment methods.

Make sure you run GettingStarted.ipynb before this one, the GettingStarted outputs will be used a inputs in this notebook.

[ ]:
```
# Load catalog from the GettingStarted notebook
gettingStarted_cat = xs.ProjectCatalog(
    str(output_folder / "example-gettingstarted.json")
)
```

Health checks

NOTE

For more information on the available `cfchecks`, `missing`, and `data_flags` methods, please consult the xclim documentation.

Health checks located under `xscen.diagnostics.health_checks` are a series of checkups that can be performed on a Dataset to make sure that it has the expected structure, frequency, calendar, etc., with the ability to call `xclim.core.cfchecks`, `xclim.core.missing`, and `xclim.core.dataflags`. The function gives full control on which checkups should raise an Exception and which should only give a UserWarning.

The arguments are:

- `structure`: Dictionary with keys “dims” and “coords” containing the expected dimensions and coordinates.
- `calendar`: Expected calendar. Synonyms should be detected correctly (e.g. “standard” and “gregorian”).
- `start_date` & `end_date`: To check if the dataset contains those.
- `variables_and_units`: Dictionary containing the expected variables and units.
- `cfchecks`: Dictionary of `xclim.core.cfchecks` to perform, per variable.
- `freq`: Expected frequency, written as the result of xr.infer_freq(ds.time).

- `missing`: String, list of strings, or dictionary of `xclim.core.missing` checks to perform.
- `flags`: Dictionary of `xclim.core.dataflags.data_flags` to perform, per variable.

Additionally, `flags_kwargs` is used to pass additional arguments to the data_flags ("dims" and "freq"), while `return_flags` can be used to return the Dataset created by `xclim.core.dataflags.data_flags`;

Use the argument `raise_on` to list to list which test should raise an error if it fails. Use ["all"] to raise on all checks.

[ ]:
```
# load input
# 'CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp245_r2i1p1f1_example-region' will be used for this␣
↪example
ds = gettingStarted_cat.search(
    id="CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp245_r2i1p1f1_example-region",
    processing_level="regridded",
).to_dataset()
```

[ ]:
```
# The checks that we want to perform. Note that all checks are optional.
structure = {"coords": ["lat", "lon", "time"]}
calendar = "365_day"  # We have a standard calendar, so we'll be warned.
start_date = "1971-01-01"  # The dataset starts on 1981-01-01, so this should fail
end_date = "2045-01-01"  # The dataset ends later, but we check that it contains at␣
↪least 2045-01-01.
variables_and_units = {"tas": "degC"}  # The dataset is in Kelvin, so we'll get warned.
cfchecks = {"tas": {"cfcheck_from_name": {}}}
freq = "MS"  # We actually have daily data, so we should get a warning.
missing = {"missing_any": {"freq": "D"}}
flags = {"tas": {"temperature_extremely_low": {}}}
```

[ ]:
```
xs.diagnostics.health_checks(
    ds,
    structure=structure,
    calendar=calendar,
    start_date=start_date,
    end_date=end_date,
    variables_and_units=variables_and_units,
    cfchecks=cfchecks,
    freq=freq,
    missing=missing,
    flags=flags,
)
```

Properties and measures

This framework for the diagnostic tests was inspired by the VALUE project. Statistical Properties is the xclim term for 'indices' in the VALUE project.

The `xscen.properties_and_measures` fonction is a wrapper for xclim.sbda.properties and xclim.sbda.measures.

- `xclim.sbda.properties` are statistical properties of a climate dataset. They allow for a better understanding of the climate by collapsing the time dimension. A few examples: mean, variance, mean spell length, annual cycle, etc.
- `xclim.sbda.measures` assess the difference between two datasets of properties. A few examples: bias, ratio, circular bias, etc.

Let's start by calculating the properties on the reference dataset. You have to provide the path to a YAML file `properties` describing the properties you want to compute. You can also specify a period to select and a unit conversion to apply before computing the properties.

This example will use a YAML file structured like this:

```
realm: generic
indicators:
  quantile_98_tas:
    base: xclim.sdba.properties.quantile
    cf_attrs:
      long_name: 98th quantile of the mean temperature
    input:
      da: tas
    parameters:
      q: 0.98
      group: time.season
  maximum_length_of_warm_spell:
    base: xclim.sdba.properties.spell_length_distribution
    cf_attrs:
      long_name: Maximum spell length distribution when the mean temperature is larger␣
↪or equal to the 90th quantile.
    input:
      da: tas
    parameters:
      method: quantile
      op: '>='
      thresh: 0.9
      stat: max
  mean-tas:
    base: xclim.sdba.properties.mean
    cf_attrs:
      long_name: Ratio of the mean temperature
    input:
      da: tas
    measure: xclim.sdba.measures.BIAS
```

[ ]:
```
properties = "samples/properties.yml"
period = [1981, 2010]
change_units_arg = {"tas": "degC"}
```

The properties can be given an argument `group` ('time', 'time.season' or 'time.month'). For 'time', the time collapsing will be performed over the whole period. For 'time.season'/'time.month', the time collapsing will be performed over each season/month. See `quantile_98_tas` as an example for season.

[ ]:
```
# load input
dref = gettingStarted_cat.search(source="ERA5-Land").to_dataset()

# calculate properties and measures
prop_ref, _ = xs.properties_and_measures(
    ds=dref,
    properties=properties,
    period=period,
    change_units_arg=change_units_arg,
```

(continues on next page)

(continued from previous page)

```
    to_level_prop="diag-properties-ref",
)

# save and update catalog
filename = str(
    output_folder
    / f"{prop_ref.attrs['cat:id']}.{prop_ref.attrs['cat:domain']}.{prop_ref.attrs['cat:
↪processing_level']}.zarr"
)
xs.save_to_zarr(ds=prop_ref, filename=filename, mode="o")
pcat.update_from_ds(ds=prop_ref, path=filename, format="zarr")
prop_ref
```

To compute a measure as well as a property, add the `dref_for_measure` argument with the reference properties calculated above. This will mesure the difference between the reference properties and the scenario properties. A default measure is associated with each properties, but it is possible to define a new one in the YAML (see `mean-tas` for example where the default (bias) was changed for ratio.)

[ ]:
```
# load input
dscen = gettingStarted_cat.search(
    source="NorESM2-MM",
    experiment="ssp245",
    member="r1.*",
    processing_level="biasadjusted",
).to_dataset()

# calculate properties and measures
prop_scen, meas_scen = xs.properties_and_measures(
    ds=dscen,
    properties=properties,
    period=period,
    dref_for_measure=prop_ref,
    change_units_arg={"tas": "degC"},
    to_level_prop="diag-properties-scen",
    to_level_meas="diag-measures-scen",
)


display(prop_scen)
display(meas_scen)

# save and update catalog
for ds in [prop_scen, meas_scen]:
    filename = str(
        output_folder
        / f"{ds.attrs['cat:id']}.{ds.attrs['cat:domain']}.{ds.attrs['cat:processing_level
↪']}.zarr"
    )
    xs.save_to_zarr(ds=ds, filename=filename, mode="o")
    pcat.update_from_ds(ds=ds, path=filename, format="zarr")
```

[ ]:
```
var = "mean-tas"
```

(continues on next page)

(continued from previous page)

```
# plot
fig, axs = plt.subplots(1, 3, figsize=(15, 5))
prop_ref[var].transpose("lat", ...).plot(ax=axs[0], cmap="plasma", vmin=3, vmax=6)
prop_scen[var].transpose("lat", ...).plot(ax=axs[1], cmap="plasma", vmin=3, vmax=6)
meas_scen[var].transpose("lat", ...).plot(ax=axs[2], cmap="RdBu_r", vmin=-3, vmax=3)
axs[0].set_title("Reference")
axs[1].set_title("Scenario")
axs[2].set_title("Bias between Reference and Scenario")
fig.tight_layout()
```

If you have different methods of bias adjustement, you might want to compare them and see for each property which method performs best (bias close to 0, ratio close to 1) with a `measures_heatmap`.

Below is an example comparing properties of a simulation (no bias adjustment) and a scenario (with quantile mapping bias adjustment). Both the simulation and the scenario use the same reference for the measures.

Note that it is possible to add many rows to `measures_heatmap`.

NOTE

The bias correction performed in the Getting Started tutorial was adjusted for speed rather than performance, using only a few quantiles. The performance results below are thus quite poor, but that was expected.

[ ]:
```
# repeat the step above for the simulation (no bias adjustment)
dsim = gettingStarted_cat.search(
    source="NorESM2-MM",
    experiment="ssp245",
    member="r1.*",
    processing_level="regridded",
).to_dataset()
prop_sim, meas_sim = xs.properties_and_measures(
    ds=dsim,
    properties=properties,
    period=period,
    dref_for_measure=prop_ref,
    change_units_arg=change_units_arg,
    to_level_prop="diag-properties-sim",
    to_level_meas="diag-measures-sim",
)

# save and update catalog
for ds in [prop_sim, meas_sim]:
    filename = str(
        output_folder
        / f"{ds.attrs['cat:id']}.{ds.attrs['cat:domain']}.{ds.attrs['cat:processing_level
↪']}.zarr"
    )
    xs.save_to_zarr(ds=ds, filename=filename, mode="o")
    pcat.update_from_ds(ds=ds, path=filename, format="zarr")
```

[ ]:
```
# load the measures for both kinds of data (sim and scen)
meas_datasets = pcat.search(
    processing_level=["diag-measures-sim", "diag-measures-scen"]
).to_dataset_dict()
```

[ ]:
```
from matplotlib import colors

# calculate the heatmap
hm = xs.diagnostics.measures_heatmap(meas_datasets=meas_datasets)

# plot the heat map
fig_hmap, ax = plt.subplots(figsize=(10, 2))
cmap = plt.cm.RdYlGn_r
norm = colors.BoundaryNorm(np.linspace(0, 1, 4), cmap.N)
im = ax.imshow(hm.heatmap.values, cmap=cmap, norm=norm)
ax.set_xticks(ticks=np.arange(3), labels=hm.properties.values, rotation=45, ha="right")
ax.set_yticks(ticks=np.arange(2), labels=hm.realization.values)
divider = make_axes_locatable(ax)
cax = divider.new_vertical(size="15%", pad=0.4)
fig_hmap.add_axes(cax)
cbar = fig_hmap.colorbar(im, cax=cax, ticks=[0, 1], orientation="horizontal")
cbar.ax.set_xticklabels(["best", "worst"])
plt.title("Normalised mean measure of properties")
fig_hmap.tight_layout()
```

`measure_improved` is another way to compare two datasets. It returns the fraction of the grid points that performed better in the second dataset than in the first dataset. It is useful to see which of properties are best corrected for by the bias adjustement method.

[ ]:
```
# change the order of meas_dataset to have sim first, because we want to see how scen␣
↪improved compared to sim.
ordered_keys = [
    "CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp245_r1i1p1f1_example-region.finer-grid.diag-
↪measures-sim.fx",
    "CMIP6_ScenarioMIP_NCC_NorESM2-MM_ssp245_r1i1p1f1_example-region.finer-grid.diag-
↪measures-scen.fx",
]
meas_datasets = {k: meas_datasets[k] for k in ordered_keys}

pb = xs.diagnostics.measures_improvement(meas_datasets=meas_datasets)

# plot
percent_better = pb.improved_grid_points.values
percent_better = np.reshape(np.array(percent_better), (1, 3))
fig_per, ax = plt.subplots(figsize=(10, 2))
cmap = plt.cm.RdYlGn
norm = colors.BoundaryNorm(np.linspace(0, 1, 100), cmap.N)
im = ax.imshow(percent_better, cmap=cmap, norm=norm)
ax.set_xticks(ticks=np.arange(3), labels=pb.properties.values, rotation=45, ha="right")
ax.set_yticks(ticks=np.arange(1), labels=[""])

divider = make_axes_locatable(ax)
cax = divider.new_vertical(size="15%", pad=0.4)
```

(continues on next page)

(continued from previous page)

```
fig_per.add_axes(cax)
cbar = fig_per.colorbar(
    im, cax=cax, ticks=np.arange(0, 1.1, 0.1), orientation="horizontal"
)
plt.title(
    "Fraction of grid cells of scen that improved or stayed the same compared to sim"
)
fig_per.tight_layout()
```

2.4.4 Ensembles

Ensemble reduction

This tutorial will explore ensemble reduction (also known as ensemble selection) using `xscen`. This will use pre-computed annual mean temperatures from `xclim.testing`.

[ ]:
```
from xclim.testing import open_dataset

import xscen as xs

datasets = {
    "ACCESS": "EnsembleStats/BCCAQv2+ANUSPLIN300_ACCESS1-0_historical+rcp45_r1i1p1_1950-
↪2100_tg_mean_YS.nc",
    "BNU-ESM": "EnsembleStats/BCCAQv2+ANUSPLIN300_BNU-ESM_historical+rcp45_r1i1p1_1950-
↪2100_tg_mean_YS.nc",
    "CCSM4-r1": "EnsembleStats/BCCAQv2+ANUSPLIN300_CCSM4_historical+rcp45_r1i1p1_1950-
↪2100_tg_mean_YS.nc",
    "CCSM4-r2": "EnsembleStats/BCCAQv2+ANUSPLIN300_CCSM4_historical+rcp45_r2i1p1_1950-
↪2100_tg_mean_YS.nc",
    "CNRM-CM5": "EnsembleStats/BCCAQv2+ANUSPLIN300_CNRM-CM5_historical+rcp45_r1i1p1_1970-
↪2050_tg_mean_YS.nc",
}

for d in datasets:
    ds = open_dataset(datasets[d]).isel(lon=slice(0, 4), lat=slice(0, 4))
    ds = xs.climatological_mean(ds, window=30, periods=[[1981, 2010], [2021, 2050]])
    datasets[d] = xs.compute_deltas(ds, reference_horizon="1981-2010")
    datasets[d].attrs["cat:id"] = d  # Required by build_reduction_data
    datasets[d].attrs["cat:xrfreq"] = "AS-JAN"
```

Preparing the data

Ensemble reduction is built upon climate indicators that are relevant to represent the ensemble's variability for a given application. In this case, we'll use the mean temperature delta between 2021-2050 and 1981-2010.

However, the functions implemented in `xclim.ensembles._reduce` require a very specific 2-D DataArray of dimensions "realization" and "criteria". That means that all the variables need to be combined and renamed, and that all dimensions need to be stacked together.

`xs.build_reduction_data` can be used to prepare the data for ensemble reduction. Its arguments are:

- `datasets` (dict, list)
- `xrfreqs` are the unique frequencies of the indicators.
- `horizons` is used to instruct on which horizon(s) to build the data from.

Because a simulation could have multiple datasets (in the case of multiple frequencies), an attempt will be made to decipher the ID and frequency from the metadata.

[ ]:
```
data = xs.build_reduction_data(
    datasets=datasets,
    xrfreqs=["AS-JAN"],
    horizons=["2021-2050"],
)

data
```

The number of criteria corresponds to: `indicators x horizons x longitude x latitude`, but criteria that are purely NaN across all realizations are removed.

Note that `xs.spatial_mean` could have been used prior to calling that function to remove the spatial dimensions.

Selecting a reduced ensemble

NOTE

Ensemble reduction in `xscen` is built upon `xclim.ensembles`. For more information on basic usage and available methods, please consult their documentation.

Ensemble reduction through `xscen.reduce_ensemble` consists in a simple call to `xclim`. The arguments are: - `data`, which is the 2D DataArray that is created by using `xs.build_reduction_data`. - `method` is either `kkz` or `kmeans`. See the link above for further details on each technique. - `kwargs` is a dictionary of arguments to send to the method chosen.

[ ]:
```
selected, clusters, fig_data = xs.reduce_ensemble(
    data=data, method="kmeans", kwargs={"method": {"n_clusters": 3}}
)
```

The method always returns 3 outputs (selected, clusters, fig_data): - `selected` is a DataArray of dimension 'realization' listing the selected simulations. - `clusters` (kmeans only) groups every realization in their respective clusters in a python dictionary. - `fig_data` (kmeans only) can be used to call `xclim.ensembles.plot_rsqprofile(fig_data)`

[ ]:
```
selected
```

[ ]:
```
# To see the clusters in more details
clusters
```

[ ]:
```
from xclim.ensembles import plot_rsqprofile

plot_rsqprofile(fig_data)
```

Ensemble partition

This tutorial will show how to use xscen to create the input for xclim partition functions.

[ ]:
```
# Get catalog
from pathlib import Path

output_folder = Path().absolute() / "_data"
cat = xs.DataCatalog(str(output_folder / "tutorial-catalog.json"))

# create a dictionnary of datasets wanted for the partition
input_dict = cat.search(variable="tas", member="r1i1p1f1").to_dataset_dict(
    xarray_open_kwargs={"engine": "h5netcdf"}
)
```

From a dictionary of datasets, the function creates a dataset with new dimensions in `partition_dim(["source", "experiment", "bias_adjust_project"]`, if they exist). In this toy example, we only have different experiments. - By default, it translates the xscen vocabulary (eg. `experiment`) to the xclim partition vocabulary (eg. `scenario`). It is possible to pass `rename_dict` to rename the dimensions with other names. - If the inputs are not on the same grid, they can be regridded through `regrid_kw` or subset to a point through `subset_kw`. The functions assumes that if there are different `bias_adjust_project`, they will be on different grids (with all `source` on the same grid). If there is one or less `bias_adjust_project`, the assumption is that`source` have different grids.

[ ]:
```
# build a single dataset
ds = xs.ensembles.build_partition_data(
    input_dict, subset_kw=dict(name="mtl", method="gridpoint", lat=[45.5], lon=[-73.6])
)
ds
```

Pass the input to an xclim partition function.

[ ]:
```
import xclim as xc

# get a yearly dataset
da = xc.atmos.tg_mean(ds=ds)

# compute uncertainty partitionning
mean, uncertainties = xc.ensembles.hawkins_sutton(da)
uncertainties
```

NOTE

Note that the figanos library provides a function `fg.partition` to plot the uncertainties.

[ ]:
```
```

2.4.5 Warming levels

This Notebook explores the options provided in `xscen` to analyze climate simulations through the scope of global warming levels, instead of temporal horizons.

First, we just need to prepare some data. We'll use NorESM2-MM as our example dataset.

[ ]:
```
# Basic imports
from pathlib import Path

import xarray as xr
import xesmf as xe
from matplotlib import pyplot as plt

import xscen as xs
from xscen.testing import datablock_3d, fake_data

# Prepare a Projectcatalog for this Tutorial.
output_folder = Path().absolute() / "_data"
project = {
    "title": "example-warminglevel",
    "description": "This is an example catalog for xscen's documentation.",
}
pcat = xs.ProjectCatalog(
    str(output_folder / "example-wl.json"),
    project=project,
    create=True,
    overwrite=True,
)

# Extract the data needed for the Tutorial
cat_sim = xs.search_data_catalogs(
    data_catalogs=[str(output_folder / "tutorial-catalog.json")],
    variables_and_freqs={"tas": "D"},
    other_search_criteria={"source": "NorESM2-MM", "activity": "ScenarioMIP"},
)
region = {
    "name": "example-region",
    "method": "bbox",
    "tile_buffer": 1.5,
    "lon_bnds": [-75, -74],
    "lat_bnds": [45, 46],
}

for ds_id, dc in cat_sim.items():
    ds = xs.extract_dataset(
        catalog=dc,
        region=region,
        xr_open_kwargs={"drop_variables": ["height", "time_bnds"]},
    )["D"]
    # Since the sample files are very small, we'll create fake data covering a longer␣
↪time period
    data = fake_data(
        nyears=121,
```

(continues on next page)

(continued from previous page)

```
        ny=len(ds.lat),
        nx=len(ds.lon),
        rand_type="tas",
        seed=list(cat_sim.keys()).index(ds_id),
        amplitude=15,
        offset=2,
    )
    attrs = ds.attrs
    ds = datablock_3d(
        data,
        "tas",
        "lon",
        -75,
        "lat",
        45,
        x_step=1,
        y_step=1.5,
        start="1/1/1981",
        freq="D",
        as_dataset=True,
    )
    ds.attrs = attrs

    filename = str(
        output_folder
        / f"wl_{ds.attrs['cat:id']}.{ds.attrs['cat:domain']}.{ds.attrs['cat:processing_
↪level']}.{ds.attrs['cat:frequency']}.zarr"
    )
    chunks = xs.io.estimate_chunks(ds, dims=["time"], target_mb=50)
    xs.save_to_zarr(ds, filename, rechunk=chunks, mode="o")
    pcat.update_from_ds(ds=ds, path=filename, info_dict={"format": "zarr"})
```

Find warming levels with only the model name

If all that you want to know is the year or the period during which a climate model reaches a given warming level, then `xs.get_warming_level` is the function to use since you can simply give it a string or a list of strings and receive that information.

The usual arguments of `xs.get_warming_level` are:

- `realization`: Dataset, dict or string.
  - Strings should follow the format 'mip-era_source_experiment_member'. Those fields should be found in the dict or in the attributes of the dataset (allowing for a possible 'cat:' prefix).
  - In all cases, regex is allowed to relax the name matching.
  - The "source" part can also be a `driving_model` name. If a `Dataset` is passed and it's `driving_model` attribute is non-null, it is used.
- `wl`: warming level.
- `window`: Number of years in the centered window during which the warming level is reached. Note that in the case of an even number, the IPCC standard is used (-n/2+1, +n/2).

- `tas_baseline_period`: The period over which the warming level is calculated, equivalent to "+0°C". Defaults to 1850-1900.
- `ignore_member`: The default `tas_src` only contains data for 1 member. If you want a result regardless of the realization number, set this to True.
- `return_horizon`: Whether to return the start/end of the horizon or to return the middle year.

It returns either a string or `['start_yr', 'end_yr']`, depending on `return_horizon`. For entries that it fails to find in the database, or for instances where a given warming level is not reached, the function returns None (or `[None, None]`).

If `realization` is a list of the accepted types, or a DataArray or a DataFrame, the function returns a sequence of the same size (and with the same index, if relevant). It can happen that a requested model's name was not found exactly in the database, but that arguments allowed for a relaxed search (`ignore_member = True` or regex in `realization`). In that case, the *selected* model doesn't have the same name as the requested one and this information is only shown in the log, unless one passes `output='selected'` to receive a dictionary instead where the keys are the *selected* models in the database.

[ ]:
```
# Multiple entries, returns a list of the same length
print(
    xs.get_warming_level(
        [
            "CMIP6_CanESM5_ssp126_r1i1p1f1",
            "CMIP6_CanESM5_ssp245_r1i1p1f1",
            "CMIP6_CanESM5_ssp370_r1i1p1f1",
            "CMIP6_CanESM5_ssp585_r1i1p1f1",
        ],
        wl=2,
        window=20,
    )
)
# Returns a list
print(
    xs.get_warming_level(
        "CMIP6_CanESM5_ssp585_r1i1p1f1", wl=2, window=20, return_horizon=True
    )
)
# Only the middle year is requested, returns a string
print(
    xs.get_warming_level(
        "CMIP6_CanESM5_ssp585_r1i1p1f1", wl=2, window=20, return_horizon=False
    )
)
# +10°C is never reached, returns None
print(xs.get_warming_level("CMIP6_CanESM5_ssp585_r1i1p1f1", wl=10, window=20))
```

Find and extract data by warming levels

If you instead need to subset and analyze data, then two options are currently provided in `xscen`: `subset_warming_level` and `produce_horizon`. - Use `subset_warming_level` when you want to cut a dataset for a period corresponding to a given warming level, but leave its frequency untouched. - Use `produce_horizon` when you need to: subset a time period, compute indicators, and compute the climatological mean for one or multiple horizons.

The two methods are detailed in the following section.

Method #1: Subsetting datasets by warming level

`xs.subset_warming_level` can be used to subset a dataset for a window over which a given global warming level is reached. A new dimension named `warminglevel` is created by the function.

The function calls `get_warming_level`, so the arguments are essentially the same.:

- `ds`: input dataset.
- `wl`: warming level.
- `window`: Number of years in the centered window during which the warming level is reached. Note that in the case of an even number, the IPCC standard is used (-n/2+1, +n/2).
- `tas_baseline_period`: The period over which the warming level is calculated, equivalent to "+0°C". Defaults to 1850-1900.
- `ignore_member`: The default database only contains data for 1 member. If you want a result regardless of the realization number, set this to True.
- `to_level`: Contrary to other methods, you can use "{wl}", "{period0}" and "{period1}" in the string to dynamically include `wl`, 'tas_baseline_period[0]' and 'tas_baseline_period[1]' in the `processing_level`.
- `wl_dim`: The string used to fill the new `warminglevel` dimension. You can use "{wl}", "{period0}" and "{period1}" in the string to dynamically include `wl`, `tas_baseline_period[0]` and `tas_baseline_period[1]`. Or you can use `True` to have a float coordinate with units of °C. If None, no new dimension will be added.

If the source, experiment, (member), and warming level are not found in the database. The function returns None.

[ ]:
```
ds = pcat.search(
    processing_level="extracted",
    experiment="ssp245",
    member="r1.*",
    source="NorESM2-MM",
    frequency="day",
).to_dataset()

xs.subset_warming_level(
    ds,
    wl=2,
    window=20,
)
```

Vectorized subsetting

The function can also vectorize the subsetting over multiple warming levels or over a properly constructed "realization" dimension. In that case, the original time axis can't be preserved. It is replaced by a fake one starting in 1000. However, as this process is a bit complex, the current xscen version only supports this if the data is annual. As the time axis doesn't carry any information, a `warminglevel_bounds` coordinate is added with the time bounds of the subsetting. If a warming level was not reached, a NaN slice is inserted in the output dataset.

This option is to be used when "scalar" subsetting is not enough, but you want to do things differently than `produce_horizons`.

Here, we'll open all experiments into a single ensemble dataset where the `realization` dimension is constructed exactly as `get_warming_level` expects it to be. We'll also average the daily data to an annual scale.

[ ]:
```
ds = pcat.search(
    processing_level="extracted",
    member="r1.*",
    frequency="day",
).to_dataset(
    # Value of the "realization" dimension will be constructed by concatenaing those␣
↪fields with a '_'
    create_ensemble_on=["mip_era", "source", "experiment", "member"]
)
ds = ds.resample(time="YS").mean()
ds
```

[ ]:
```
xs.subset_warming_level(ds, wl=[1.5, 2, 3], wl_dim=True, to_level="warming-level")
```

Method #2: Producing horizons

If what you need is to compute indicators and their climatological mean, `xs.aggregate.produce_horizon` is a more convenient function to work with than `subset_warming_level`. Since the years are meaningless for warming levels, and are even detrimental to making ensemble statistics, the function formats the output as to remove the 'time' and 'year' information from the dataset, while the seasons/months are unstacked to different coordinates. Hence, the single dataset outputed can contain indicators of different frequencies, as well as multiple warming levels or temporal horizons.

The arguments of `xs.aggregate.produce_horizon` are:

- `ds`: input dataset.
- `indicators`: As in `compute_indicators`
- `periods`: Periods to cut.
- `warminglevels`: Dictionary of arguments to pass to `subset_warming_level`. If 'wl' is a list, the function will be called for each value and produce multiple horizons.

If both periods and warminglevels are None, the full time series will be used. If a dataset does not contain a given period or warming level, then that specific period will be skipped.

[ ]:
```
dict_input = pcat.search(processing_level="extracted", xrfreq="D").to_dataset_dict(
    xarray_open_kwargs={"decode_timedelta": False}
)
```

(continues on next page)

(continued from previous page)

```
for id_input, ds_input in dict_input.items():
    # 1981-2010 will be used as our reference period. We can compute it at the same time.
    ds_hor = xs.produce_horizon(
        ds_input,
        indicators="samples/indicators.yml",
        periods=[["1981", "2010"]],
        warminglevels={"wl": [1, 1.5, 2], "window": 30, "ignore_member": True},
        to_level="horizons",
    )

    # Save
    filename = str(
        output_folder
        / f"wl_{ds_hor.attrs['cat:id']}.{ds_hor.attrs['cat:domain']}.{ds_hor.attrs['cat:
↪processing_level']}.{ds_hor.attrs['cat:frequency']}.zarr"
    )
    xs.save_to_zarr(ds_hor, filename, mode="o")
    pcat.update_from_ds(ds=ds_hor, path=filename, info_dict={"format": "zarr"})
```

[ ]:
```
display(ds_hor)
```

Deltas and spatial aggregation

This step is done as in the *Getting Started* Notebook. Here we will spatially aggregate the data, but the datasets could also be regridded to a common grid.

[ ]:
```
dict_wl = pcat.search(processing_level="horizons").to_dataset_dict(
    xarray_open_kwargs={"decode_timedelta": False}
)

for id_wl, ds_wl in dict_wl.items():
    # compute delta
    ds_delta = xs.aggregate.compute_deltas(
        ds=ds_wl, reference_horizon="1981-2010", to_level="deltas"
    )

    # remove the reference period from the dataset
    ds_delta = ds_delta.sel(horizon=~ds_delta.horizon.isin(["1981-2010"]))

    # aggregate
    ds_delta["lon"].attrs["axis"] = "X"
    ds_delta["lat"].attrs["axis"] = "Y"
    ds_agg = xs.spatial_mean(
        ds_delta,
        method="cos-lat",
        to_level="deltas-agg",
    )

    # Save
    filename = str(
        output_folder
```

(continues on next page)

(continued from previous page)

```
        / f"wl_{ds_agg.attrs['cat:id']}.{ds_agg.attrs['cat:domain']}.{ds_agg.attrs['cat:
↪processing_level']}.{ds_agg.attrs['cat:frequency']}.zarr"
    )
    xs.save_to_zarr(ds_agg, filename, mode="o")
    pcat.update_from_ds(ds=ds_agg, path=filename, info_dict={"format": "zarr"})
```

[ ]:
```
display(ds_agg)
```

Ensemble statistics

Even more than with time-based horizons, the first step of ensemble statistics should be to generate the weights. Indeed, if a model has 3 experiments reaching a given warming level, we want it to have the same weight as a model with only 2 experiments reaching that warming level. The argument `skipna=False` should be passed to `xs.generate_weights` to properly assess which simulations reaches which warming level. If the `horizon` dimension differs between datasets (as is the case here), they are reindexed and given a weight of 0.

When working with warming levels, how to assess experiments is more open-ended. The IPCC Atlas splits the statistics and climate change signals by SSPs, even when they are being analyzed through warming levels, but experiments could also be considered as 'members' of a given model and used to bolster the number of realizations.

[ ]:
```
datasets = pcat.search(processing_level="deltas-agg").to_dataset_dict()
```

[ ]:
```
# All realisations of NorESM2-MM are given the same weight for the first horizon, while␣
↪it correctly recognizes that +1.5°C and +2°C weren't reached for the SSP126.
weights = xs.ensembles.generate_weights(
    datasets=datasets, independence_level="model", skipna=False
)
display(weights)
```

Next, the weights and the datasets can be passed to `xs.ensemble_stats` to calculate the ensemble statistics.

[ ]:
```
ds_ens = xs.ensemble_stats(
    datasets=datasets,
    common_attrs_only=True,
    weights=weights,
    statistics={"ensemble_mean_std_max_min": None},
    to_level=f"ensemble-deltas-wl",
)

# It is sometimes useful to keep track of how many realisations made the ensemble.
ds_ens.horizon.attrs["ensemble_size"] = len(datasets)

filename = str(
    output_folder
    / f"wl_{ds_ens.attrs['cat:id']}.{ds_ens.attrs['cat:domain']}.{ds_ens.attrs['cat:
↪processing_level']}.{ds_ens.attrs['cat:frequency']}.zarr"
)
xs.save_to_zarr(ds_ens, filename, mode="o")
pcat.update_from_ds(ds=ds_ens, path=filename, info_dict={"format": "zarr"})
```

[ ]:
```
display(ds_ens)
```

2.4.6 YAML usage

NOTE

This tutorial will mostly remain xscen-specific and, thus, will not go into more advanced YAML functionalities such as anchors. More information on that can be consulted here, while this template makes ample use of them.

While parameters can be explicitely given to functions, most support the use of YAML configuration files to automatically pass arguments. This tutorial will go over basic principles on how to write and prepare configuration files, and provide a few examples.

An `xscen` function supports YAML parametrisation if it is preceded by the `parse_config` wrapper in the code. Currently supported functions are:

[ ]:
```
from xscen.config import get_configurable

list(get_configurable().keys())
```

Loading an existing YAML config file

YAML files are read using `xscen.load_config`. Any number of files can be called, which will be merged together into a single python dictionary accessed through `xscen.CONFIG`.

[ ]:
```
from pathlib import Path

import xscen as xs
from xscen import CONFIG
```

[ ]:
```
# Load configuration
xs.load_config(
    str(
        Path().absolute().parent.parent
        / "templates"
        / "1-basic_workflow_with_config"
        / "config1.yml"
    ),
    # str(Path().absolute().parent.parent / "templates" / "1-basic_workflow_with_config"␣
↪/ "paths1_example.yml")  We can't actually load this file due to the fake paths, but␣
↪this would be the format
)

# Display the dictionary keys
print(CONFIG.keys())
```

`xscen.CONFIG` behaves similarly to a python dictionary, but has a custom `__getitem__` that returns a `deepcopy` of the requested item. As such, it is unmutable and thus, reliable and robust.

```
[ ]: # A normal python dictionary is mutable, but a CONFIG dictionary is not.
pydict = dict(CONFIG["project"])
print(CONFIG["project"]["id"], ", ", pydict["id"])
pydict2 = pydict
pydict2["id"] = "modified id"
print(CONFIG["project"]["id"], ", ", pydict["id"], ", ", pydict2["id"])
pydict3 = pydict2
pydict3["id"] = "even more modified id"
print(
    CONFIG["project"]["id"],
    ", ",
    pydict["id"],
    ", ",
    pydict2["id"],
    ", ",
    pydict3["id"],
)
```

If one really want to modify the `CONFIG` dictionary from within the workflow itself, its `set` method must be used.

```
[ ]: CONFIG.set("project.id", "modified id")
print(CONFIG["project"]["id"])
```

Building a YAML config file

Generic arguments

Since `CONFIG` is a python dictionary, anything can be written in it if it is deemed useful for the execution of the script. A good practice, such as seen in this template's config1.yml, is for example to use the YAML file to provide a list of tasks to be accomplished, give the general description of the project, or provide a dask configuration:

```
[ ]: print(CONFIG["tasks"])
print(CONFIG["project"])
print(CONFIG["regrid"]["dask"])
```

These are not linked to any function and will not automatically be called upon by `xscen`, but can be referred to during the execution of the script. Below is an example where `tasks` is used to instruct on which tasks to accomplish and which to skip. Many such example can be seen throughout the provided templates.

```
[ ]: if "extract" in CONFIG["tasks"]:
    print("This will start the extraction process.")

if "figures" in CONFIG["tasks"]:
    print(
        "This would start creating figures, but it will be skipped since it is not in␣
↪the list of tasks."
    )
```

Function-specific parameters

In addition to generic arguments, a major convenience of YAML files is that parameters can be automatically fed to functions if they are wrapped by `@parse_config` (see above for the list of currently supported functions). The exact following format has to be used:

```
module:
    function:
        argument:
```

The most up-to-date list of modules can be consulted here, as well as at the start of this tutorial. A simple example would be as follows:

```
aggregate:
  compute_deltas:
    kind: "+"
    reference_horizon: "1991-2020"
    to_level: 'delta'
```

Some functions have arguments in the form of lists and dictionaries. These are also supported:

```
extract:
    search_data_catalogs:
      variables_and_freqs:
        tasmax: D
        tasmin: D
        pr: D
        dtr: D
      allow_resampling: False
      allow_conversion: True
      periods: ['1991', '2020']
      other_search_criteria:
        source:
          "ERA5-Land"
```

[ ]:
```
# Note that the YAML used here is more complex and separates tasks between 'reconstruction
↪' and 'simulation', which would break the automatic passing of arguments.
print(
    CONFIG["extract"]["reconstruction"]["search_data_catalogs"]["variables_and_freqs"]
)  # Dictionary
print(CONFIG["extract"]["reconstruction"]["search_data_catalogs"]["periods"])  # List
```

Let's test that it is working, using `climatological_op`:

[ ]:
```
# We should obtain 30-year means separated in 10-year intervals.
CONFIG["aggregate"]["climatological_op"]
```

[ ]:
```
import pandas as pd
import xarray as xr

# Create a dummy dataset
time = pd.date_range("1951-01-01", "2100-01-01", freq="AS-JAN")
da = xr.DataArray([0] * len(time), coords={"time": time})
```

(continues on next page)

(continued from previous page)

```
da.name = "test"
ds = da.to_dataset()

# Call climatological_op using no argument other than what's in CONFIG
print(xs.climatological_op(ds))
```

Managing paths

As a final note, it should be said that YAML files are a good way to privately provide paths to a script without having to explicitely write them in the code. An example is provided here. As stated earlier, `xs.load_config` will merge together the provided YAML files into a single dictionary, meaning that the separation will be seamless once the script is running.

As an added protection, if the script is to be hosted on Github, `paths.yml` (or whatever it is being called) can then be added to the `.gitignore`.

Configuration of external packages

As explained in the `load_config` documentation, a few top-level sections can be used to configure packages external to xscen. For example, everything under the `logging` section will be sent to `logging.config.dictConfig(...)`, allowing the full configuration of python's built-in logging mechanism. The current config does exactly that by configuring a logger for `xscen` that logs to the console, with a sensibility set to the INFO level and a specified record formating :

[ ]:
```
CONFIG["logging"]
```

Passing configuration through the command line

In order to have a more flexible configuration, it can be interesting to modify it using the command line. This way, the workflow can be started with different values without having to edit and save the YAML file each time. Alternatively, the command line arguments can also be used to determine which configuration file to use, so that the same workflow can be launched with different configurations without needing to duplicate the code. The second template workflow uses this method.

The idea is simply to create an `ArgumentParser` with python's built-in argparse :

[ ]:
```
from argparse import ArgumentParser

parser = ArgumentParser(description="An example CLI arguments parser.")
parser.add_argument("-c", "--conf", action="append")

# Let's simulate command line arguments
example_args = (
    "-c ../../templates/2-indicators_only/config2.yml "
    '-c project.title="Title" '
    "--conf project.id=newID"
)

args = parser.parse_args(example_args.split())
print(args.conf)
```

And then we can simply pass this list to `load_config`, which accepts file paths and "key=value" pairs.

```
[ ]: xs.load_config(*args.conf)

print(CONFIG["project"]["title"])
print(CONFIG["project"]["id"])
```

2.5 Columns

This section presents a definition and examples for each column of a xscen DataCatalog. The entries for the columns are based on CMIP6 metadata and the ES-DOC controlled vocabulary (https://github.com/ES-DOC/pyessv-archive). Some columns might be left empty (with a `NaN`), but `id`, `domain`, `processing_level` and `xrfreq` are mandatory. These four columns are what xscen uses by default to guess which entries can be merged together : all entries with the same unique combination of the four columns will be combined in a single Dataset when any of the first three functions listed *here* are used.

- **`id`: Unique Dataset ID generated by `xscen` based on a subset of columns. By default, it is based on `xscen.catalog.ID_COLUMNS`.**
  - E.g. "ERA_ecmwf_ERA5_ERA5-Land_NAM", "CMIP6_ScenarioMIP_CMCC_CMCC-ESM2_ssp245_r1i1p1f1_global"
- **`type`: Type of data.**
  - E.g. "forecast", "station-obs", "gridded-obs", "reconstruction", "simulation"
- **`processing_level`: Level of post-processing reached.**
  - E.g. "raw", "extracted", "regridded", "biasadjusted"
- **`bias_adjust_institution`: Institution that computed the bias adjustment.**
  - E.g. "Ouranos", "PCIC"
- **`bias_adjust_project`: Name of the project that computed the bias adjustment.**
  - E.g. "ESPO-R5", "BCCAQv2"
- **`mip_era`: CMIP generation associated with the data.**
  - E.g. "CMIP6", "CMIP5"
- **`activity`: Model Intercomparison Project (MIP) associated with the data. This is the same as *activity_id* in CMIP6 data. CMIP is the activity for the historical experiment and the DECK experiments.**
  - E.g. "CMIP", "CORDEX", "HighResMIP"
- **`driving_model`: Name of the driver. Following the *driving_model* convention from ES-DOC, this is in the format "institution-model".**
  - E.g. "CCCma-CanESM2"
- **`institution`: Institution associated with the source.**
  - E.g. "CCCma", "Ouranos", "ECMWF"
- **`source`: For simulation type, this is the model. For GCMs, this is the name of the model (*source_id* in CMIP6 and *rcm_name* in ES-DOC for CORDEX). For reconstruction type, this is the name of the dataset.**
  - E.g. "CanESM5", "CRCM5", "ERA5", "ERA5-Land, ERA5-Preliminary"
- **`experiment`: Name of the experiment of the model.**

  - E.g. “historical”, “ssp245”, “rcp85”
- **`member`: Name of the realisation. For RCMs, this is the member associated with the driver.**
  - E.g. “r1i1p1f1”
- **`xrfreq`: Pandas/xarray frequency.**
  - E.g. “YS”, “QS-DEC”
- **`frequency`: Frequency in letters (CMIP6 format).**
  - E.g. “yr”,”qtr”
- **`variable`: Variables in the dataset. It can be a Tuple.**
  - E.g. “tasmax”, (“tasmax”, “tasmin”, “pr”)
- **`domain`: Name of the region covered by the dataset. It can also contain information on the grid.**
  - E.g. “global”, “NAM”, “ARC-44”, “ARC-22”
- **`date_start`: First date of the dataset. This usually is a Datetime object with a ms resolution.**
  - E.g. “2022-06-03 00:00:00”
- **`date_end`: Last date of the dataset. This usually is a Datetime object with a ms resolution.**
  - E.g. “2022-06-03 00:00:00”
- **`version`: Version of the dataset.**
  - E.g. “1.0”
- **`format`: Format of the dataset.**
  - E.g. “zarr”, “nc”
- **`path`: Path to the dataset.**
  - E.g. “/some/path/to/the/data.zarr”

2.6 Workflow templates

This folder contains templates of xscen workflows to provide additional “real-world” examples besides the notebooks and API docs. Most of them are not usable as-is, but usually only the configuration (some yaml files) needs to be edited.

Most of the templates are heavily commented in their python code as well as their configuration files, but a small summary of what each does is included here.

Warning: The link above brings you to the development version of the templates. You might want to access a version specific to your installed xscen. Click the dropdown menu in the upper left corner that says “main” and navigate to “tags” and the specific version of interest.

2.6.1 1 - Basic workflow with config

The archetypal xscen workflow that does every steps of a normal climate scenarisation project : extract, regrid, biasadjust, cleanup, rechunk, diagnostics, indicators, climatology, delta, ensembles. It is controlled from the `config1.yml` file. For each step, it will iterate over each member of the ensemble, thus creating many intermediate files before the final products.

2.6.2 2 - Compute indicators

A basic, single-step workflow to compute a list (module) of xclim indicators. Also controlled from its `config2.yml` file, but its path needs to be passed to the script through the command line.

2.7 API

2.7.1 Catalog

Catalog objects and related tools.

xscen.catalog.COLUMNS = ['id', 'type', 'processing_level', 'bias_adjust_institution', 'bias_adjust_project', 'mip_era', 'activity', 'driving_model', 'institution', 'source', 'experiment', 'member', 'xrfreq', 'frequency', 'variable', 'domain', 'date_start', 'date_end', 'version', 'format', 'path']

Official column names.

class xscen.catalog.**DataCatalog**(**args*, ***kwargs*)

A read-only intake_esm catalog adapted to xscen's syntax.

This class expects the catalog to have the columns listed in *xscen.catalog.COLUMNS* and it comes with default arguments for reading the CSV files (`xscen.catalog.csv_kwargs`). For example, all string columns (except *path*) are cast to a categorical dtype and the datetime columns are parsed with a special function that allows dates outside the conventional *datetime64[ns]* bounds by storing the data using `pandas.Period` objects.

Parameters

- ***args** (*str or os.PathLike or dict*) – Path to a catalog JSON file. If a dict, it must have two keys: 'esmcat' and 'df'. 'esmcat' must be a dict representation of the ESM catalog. 'df' must be a Pandas DataFrame containing content that would otherwise be in the CSV file.
- **check_valid** (*bool*) – If True, will check that all files in the catalog exist on disk and remove those that don't.
- **drop_duplicates** (*bool*) – If True, will drop duplicates in the catalog based on the 'id' and 'path' columns.
- ****kwargs** (*dict*) – Any other arguments are passed to intake_esm.esm_datastore.

See also:

`intake_esm.core.esm_datastore`

check_valid()

Verify that all files in the catalog exist on disk and remove those that don't.

If a file is a Zarr, it will also check that all variables are present and remove those that aren't.

`drop_duplicates`(*columns: list[str] | None = None*)

Drop duplicates in the catalog based on a subset of columns.

Parameters
columns (*list of str, optional*) – The columns used to identify duplicates. If None, 'id' and 'path' are used.

`exists_in_cat`(***columns*) → bool

Check if there is an entry in the catalogue corresponding to the arguments given.

Parameters
columns (Arguments that will be given to *catalog.search*)

Returns
bool – True if there is an entry in the catalogue corresponding to the arguments given.

`classmethod from_df`(*data: DataFrame | PathLike | Sequence[PathLike]*, *esmdata: PathLike | dict | None = None*, ***, *read_csv_kwargs: Mapping[str, Any] | None = None*, *name: str = 'virtual'*, ***intake_kwargs*)

Create a DataCatalog from one or more csv files.

Parameters

- **data** (*DataFrame or path or sequence of paths*) – A DataFrame or one or more paths to csv files.
- **esmdata** (*path or dict, optional*) – The "ESM collection data" as a path to a json file or a dict. If None (default), xscen's default `esm_col_data` is used.
- **read_csv_kwargs** (*dict, optional*) – Extra kwargs to pass to *pd.read_csv*, in addition to the ones in `csv_kwargs`.
- **name** (*str*) – If *metadata* doesn't contain it, a name to give to the catalog.

See also:

`pandas.read_csv`

`iter_unique`(**columns*)

Iterate over sub-catalogs for each group of unique values for all specified columns.

This is a generator that yields a tuple of the unique values of the current group, in the same order as the arguments, and the sub-catalog.

`search`(***columns*)

Modification of .search() to add the 'periods' keyword.

`to_dataset`(*concat_on: str | list[str] | None = None*, *create_ensemble_on: str | list[str] | None = None*, *ensemble_name: list[str] | None = None*, *calendar: str | None = 'standard'*, ***kwargs*) → Dataset

Open the catalog's entries into a single dataset.

Same as `to_dask()`, but with additional control over the aggregations. The dataset definition logic is left untouched by this method (by default: ["id", "domain", "processing_level", "xrfreq"]), except that newly aggregated columns are removed from the "id". This will override any "custom" id, ones not unstackable with *unstack_id()*.

Ensemble preprocessing logic is taken from `xclim.ensembles.create_ensemble()`. When *create_ensemble_on* is given, the function ensures all entries have the correct time coordinate according to *xrfreq*.

Parameters

- **concat_on** (*list of str or str, optional*) – A list of catalog columns over which to concat the datasets (in addition to 'time'). Each will become a new dimension with the column values as coordinates. Xarray concatenation rules apply and can be acted upon through *xarray_combine_by_coords_kwargs*.
- **create_ensemble_on** (*list of str or str, optional*) – The given column values will be merged into a new id-like "realization" column, which will be concatenated over. The given columns are removed from the dataset id, to remove them from the groupby_attrs logic. Xarray concatenation rules apply and can be acted upon through *xarray_combine_by_coords_kwargs*.
- **ensemble_name** (*list of strings, optional*) – If *create_ensemble_on* is given, this can be a subset of those column names to use when constructing the realization coordinate. If None, this will be the same as *create_ensemble_on*. The resulting coordinate must be unique.
- **calendar** (*str, optional*) – If *create_ensemble_on* is given, all datasets are converted to this calendar before concatenation. Ignored otherwise (default). If None, no conversion is done. *align_on* is always "date".
- **kwargs** – Any other arguments are passed to `to_dataset_dict()`. The *preprocess* argument cannot be used if *create_ensemble_on* is given.

Returns

`Dataset`

See also:

`intake_esm.core.esm_datastore.to_dataset_dict`, `intake_esm.core.esm_datastore.to_dask`, `xclim.ensembles.create_ensemble`

unique(*columns: str | Sequence[str] | None = None*)

Return a series of unique values in the catalog.

Parameters

columns (*str or sequence of str, optional*) – The columns to get unique values from. If None, all columns are used.

xscen.catalog.**ID_COLUMNS = ['bias_adjust_project', 'mip_era', 'activity', 'driving_model', 'institution', 'source', 'experiment', 'member', 'domain']**

Default columns used to create a unique ID

class xscen.catalog.**ProjectCatalog**(**args*, ***kwargs*)

A DataCatalog with additional 'write' functionalities that can update and upload itself.

See also:

`intake_esm.core.esm_datastore`

classmethod create(*filename: PathLike | str*, ***, *project: dict | None = None*, *overwrite: bool = False*)

Create a new project catalog from some project metadata.

Creates the json from default `esm_col_data` and an empty csv file.

Parameters

- **filename** (*os.PathLike or str*) – A path to the json file (with or without suffix).
- **project** (*dict, optional*) – Metadata to create the catalog. If None, *CONFIG['project']* will be used. Valid fields are:
  - title : Name of the project, given as the catalog's "title".

- **id**
  [slug-like version of the name, given as the catalog's id (should be url-proof)] Defaults to a modified name.
- version : Version of the project (and thus the catalog), string like "x.y.z".
- description : Detailed description of the project, given to the catalog's "description".
- Any other entry defined in `esm_col_data`.

At least one of *id* and *title* must be given, the rest is optional.

- **overwrite** (*bool*) – If True, will overwrite any existing JSON and CSV file.

Returns
ProjectCatalog – An empty intake_esm catalog.

`refresh`()

Reread the catalog CSV saved on disk.

`update`(*df:* DataCatalog | *esm_datastore* | *DataFrame* | *Series* | *Sequence[Series]* | *None = None*)

Update the catalog with new data and writes the new data to the csv file.

Once the internal dataframe is updated with *df*, the csv on disk is parsed, updated with the internal dataframe, duplicates are dropped and everything is written back to the csv. This means that nothing is _removed_* from the csv when calling this method, and it is safe to use even with a subset of the catalog.

> **Warning:** If a file was deleted between the parsing of the catalog and this call, it will be removed from the csv when *check_valid* is called.

Parameters
df (*Union[DataCatalog, intake_esm.esm_datastore, pd.DataFrame, pd.Series, Sequence[pd.Series]], optional*) – Data to be added to the catalog. If None, nothing is added, but the catalog is still updated.

`update_from_ds`(*ds:* Dataset, *path: PathLike | str*, *info_dict: dict | None = None*, ***info_kwargs*)

Update the catalog with new data and writes the new data to the csv file.

We get the new data from the attributes of *ds*, the dictionary *info_dict* and *path*.

Once the internal dataframe is updated with the new data, the csv on disk is parsed, updated with the internal dataframe, duplicates are dropped and everything is written back to the csv. This means that nothing is _removed_* from the csv when calling this method, and it is safe to use even with a subset of the catalog.

> **Warning:** If a file was deleted between the parsing of the catalog and this call, it will be removed from the csv when *check_valid* is called.

Parameters

- **ds** (*xarray.Dataset*) – Dataset that we want to add to the catalog. The columns of the catalog will be filled from the global attributes starting with 'cat:' of the dataset.
- **info_dict** (*dict, optional*) – Extra information to fill in the catalog.
- **path** (*os.PathLike or str*) – Path to the file that contains the dataset. This will be added to the 'path' column of the catalog.

xscen.catalog.**concat_data_catalogs**(**dcs*)

Concatenate a multiple DataCatalogs.

Output catalog is the union of all rows and all derived variables, with the "esmcat" of the first DataCatalog. Duplicate rows are dropped and the index is reset.

xscen.catalog.**generate_id**(*df: DataFrame | Dataset*, *id_columns: list | None = None*) → Series

Create an ID from column entries.

Parameters

- **df** (*pd.DataFrame, xr.Dataset*) – Data for which to create an ID.
- **id_columns** (*list, optional*) – List of column names on which to base the dataset definition. Empty columns will be skipped. If None (default), uses *ID_COLUMNS*.

Returns

pd.Series – A series of IDs, one per row of the input DataFrame.

xscen.catalog.**unstack_id**(*df: DataFrame | ProjectCatalog | DataCatalog*) → dict

Reverse-engineer an ID using catalog entries.

Parameters

df (*Union[pd.DataFrame, ProjectCatalog, DataCatalog]*) – Either a Project/DataCatalog or a pandas DataFrame.

Returns

dict – Dictionary with one entry per unique ID, which are themselves dictionaries of all the individual parts of the ID.

Catalog creation and path building tools.

xscen.catutils.**build_path**(*data: dict | Dataset | DataArray | Series | DataCatalog | DataFrame*, *schemas: str | PathLike | dict | None = None*, *root: str | PathLike | None = None*, ***extra_facets*) → Path | *DataCatalog* | DataFrame

Parse the schema from a configuration and construct path using a dictionary of facets.

Parameters

- **data** (*dict or xr.Dataset or xr.DataArray or pd.Series or DataCatalog or pd.DataFrame*) – Dict of facets. Or xarray object to read the facets from. In the latter case, variable and time-dependent facets are read with *parse_from_ds()* and supplemented with all the object's attribute, giving priority to the "official" xscen attributes (prefixed with *cat:*, see *xscen.utils.get_cat_attrs()*). Can also be a catalog or a DataFrame, in which a "new_path" column is generated for each item.
- **schemas** (*Path or dict, optional*) – Path to YAML schematic of database schema. If None, will use a default schema. See the comments in the *xscen/data/file_schema.yml* file for more details on its construction. A dict of dict schemas can be given (same as reading the yaml). Or a single schema dict (single element of the yaml).
- **root** (*str or Path, optional*) – If given, the generated path(s) is given under this root one.
- ****extra_facets** – Extra facets to supplement or override metadadata missing from the first input.

Returns

Path or catalog – Constructed path. If "format" is absent from the facets, it has no suffix. If *data* was a catalog, a copy with a "new_path" column is returned. Another "new_path_type" column is also added if *schemas* was a collection of schemas (like the default).

Examples

To rename a full catalog, the simplest way is to do:

```
>>> import xscen as xs
>>> import shutil as sh
>>> new_cat = xs.catutils.build_path(old_cat)
>>> for i, row in new_cat.iterrows():
...     sh.move(row.path, row.new_path)
...
```

xscen.catutils.**parse_directory**(*directories: list[str | PathLike]*, *patterns: list[str]*, ***, *id_columns: list[str] | None = None*, *read_from_file: bool | Sequence[str] | tuple[Sequence[str], Sequence[str]] | Sequence[tuple[Sequence[str], Sequence[str]]] = False*, *homogenous_info: dict | None = None*, *cvs: str | PathLike | dict | None = None*, *dirglob: str | None = None*, *xr_open_kwargs: Mapping[str, Any] | None = None*, *only_official_columns: bool = True*, *progress: bool = False*, *parallel_dirs: bool | int = False*, *file_checks: list[str] | None = None*) → DataFrame

Parse files in a directory and return them as a pd.DataFrame.

Parameters

- **directories** (*list of os.PathLike or list of str*) – List of directories to parse. The parse is recursive.
- **patterns** (*list of str*) – List of possible patterns to be used by `parse.parse()` to decode the file names. See Notes below.
- **id_columns** (*list of str, optional*) – List of column names on which to base the dataset definition. Empty columns will be skipped. If None (default), it uses `ID_COLUMNS`.
- **read_from_file** (*boolean or set of strings or tuple of 2 sets of strings or list of tuples*) – If True, if some fields were not parsed from their path, files are opened and missing fields are parsed from their metadata, if found. If a sequence of column names, only those fields are parsed from the file, if missing. If False (default), files are never opened. If a tuple of 2 lists of strings, only the first file of groups defined by the first list of columns is read and the second list of columns is parsed from the file and applied to the whole group. For example, *([“source”],[“institution”, “activity”])* will find a group with all the files that have the same source, open only one of the files to read the institution and activity, and write this information in the catalog for all filles of the group. It can also be a list of those tuples.
- **homogenous_info** (*dict, optional*) – Using the {column_name: description} format, information to apply to all files. These are applied before the *cvs*.
- **cvs** (*str or os.PathLike or dict, optional*) – Dictionary with mapping from parsed term to preferred terms (Controlled VocabularieS) for each column. May have an additional “attributes” entry which maps from attribute names in the files to official column names. The attribute translation is done before the rest. In the “variable” entry, if a name is mapped to None (null), that variable will not be listed in the catalog. A term can map to another mapping from field name to values, so that a value on one column triggers the filling of other columns. In the latter case, that other column must exist beforehand, whether it was in the pattern or in the homogenous_info.
- **dirglob** (*str, optional*) – A glob pattern for path matching to accelerate the parsing of a directory tree if only a subtree is needed. Only folders matching the pattern are parsed to find datasets.

- **xr_open_kwargs** (*dict*) – If needed, arguments to send xr.open_dataset() when opening the file to read the attributes.
- **only_official_columns** (*bool*) – If True (default), this ensures the final catalog only has the columns defined in `xscen.catalog.COLUMNS`. Other fields in the patterns will raise an error. If False, the columns are those used in the patterns and the homogenous info. In that case, the column order is not determined. Path, format and id are always present in the output.
- **progress** (*bool*) – If True, a counter is shown in stdout when finding files on disk. Does nothing if *parallel_dirs* is not False.
- **parallel_dirs** (*bool or int*) – If True, each directory is searched in parallel. If an int, it is the number of parallel searches. This should only be significantly useful if the directories are on different disks.
- **file_checks** (*list of str, optional*) – A list of file checks to run on the parsed files. Available values are: - "readable" : Check that the file is readable by the current user. - "writable" : Check that the file is writable by the current user. - "ncvalid" : For netCDF, check that it is valid (openable with netCDF4). Any check will slow down the parsing.

Notes

- **Offical columns names are controlled and ordered by `COLUMNS`:**

  ["id", "type", "processing_level", "mip_era", "activity", "driving_institution", "driving_model", "institution",
  "source", "bias_adjust_institution", "bias_adjust_project","experiment", "member", "xrfreq", "frequency", "variable", "domain", "date_start", "date_end", "version"]

- **Not all column names have to be present, but "xrfreq" (obtainable through "frequency"), "variable",**
  "date_start" and "processing_level" are necessary for a workable catalog.

- **'patterns' should highlight the columns with braces.**
  This acts like the reverse operation of *format()*. It is a template string with *{field name:type}* elements. The default "type" will match alphanumeric parts of the path, excluding the "_", "/" and "" characters. The "_" type will allow underscores. Field names prefixed by "?" will not be included in the output. See the documentation of `parse` for more type options. You can also add your own types using the `register_parse_type()` decorator.

  The "DATES" field is special as it will only match dates, either as a single date (YYYY, YYYYMM, YYYYMMDD) assigned to "{date_start}" (with "date_end" automatically inferred) or two dates of the same format as "{date_start}-{date_end}".

  Example: *"{source}/{?ignored project name}_{?:_}_{DATES}.nc"* Here, "source" will be the full folder name and it can't include underscores. The first section of the filename will be excluded from the output, it was given a name (ignore project name) to make the pattern readable. The last section of the filenames ("dates") will yield a "date_start" / "date_end" couple. All other sections in the middle will be ignored, as they match "{?:_}".

Returns
pd.DataFrame – Parsed directory files

`xscen.catutils.parse_from_ds`(*obj: str | PathLike | Dataset*, *names: Sequence[str]*, *attrs_map: Mapping[str, str] | None = None*, ***xrkwargs*)

Parse a list of catalog fields from the file/dataset itself.

If passed a path, this opens the file.

Infers the variable from the variables. Infers xrfreq, frequency, date_start and date_end from the time coordinate if present. Infers other attributes from the coordinates or the global attributes. Attributes names can be translated using the *attrs_map* mapping (from file attribute name to name in *names*).

If the obj is the path to a Zarr dataset and none of "frequency", "xrfreq", "date_start" or "date_end" are requested, `parse_from_zarr()` is used instead of opening the file.

Parameters

- **obj** (*str or os.PathLike or xr.Dataset*) – Dataset to parse.
- **names** (*sequence of str*) – List of attributes to be parsed from the dataset.
- **attrs_map** (*dict, optional*) – In the case of non-standard names in the file, this can be used to match entries in the files to specific 'names' in the requested list.
- **xrkwargs** – Arguments to be passed to open_dataset().

`xscen.catutils.register_parse_type`(*name: str*, *regex: str = '([^_\\/\\\\]*)'*, *group_count: int = 1*)

Register a new parse type to be available in *parse_directory()* patterns.

Function decorated by this will be registered in `EXTRA_PARSE_TYPES`. The function must take a single string and should return a single string. If you return a different type, it may interfere with the other steps of *parse_directory*.

Parameters

- **name** (*str*) – The type name. To make use of this type, put "{field:name}" in your pattern.
- **regex** (*str*) – A regex string to determine what can be matched by this type. The default matches anything but / and _, same as the default parse type.
- **group_count** (*int*) – The number of regex groups in the previous regex string.

2.7.2 Extraction

Functions to find and extract data from a catalog.

`xscen.extract.extract_dataset`(*catalog:* DataCatalog, *, *variables_and_freqs: dict | None = None*, *periods: list[str] | list[list[str]] | None = None*, *region: dict | None = None*, *to_level: str = 'extracted'*, *ensure_correct_time: bool = True*, *xr_open_kwargs: dict | None = None*, *xr_combine_kwargs: dict | None = None*, *preprocess: Callable | None = None*, *resample_methods: dict | None = None*, *mask: bool | Dataset | DataArray = False*) → dict

Take one element of the output of *search_data_catalogs* and returns a dataset, performing conversions and resampling as needed.

Nothing is written to disk within this function.

Parameters

- **catalog** (*DataCatalog*) – Sub-catalog for a single dataset, one value of the output of *search_data_catalogs*.
- **variables_and_freqs** (*dict, optional*) – Variables and freqs, following a 'variable: xrfreq-compatible str' format. A list of strings can also be provided. If None, it will be read from catalog._requested_variables and catalog._requested_variable_freqs (set by *variables_and_freqs* in *search_data_catalogs*)
- **periods** (*list of str or list of lists of str, optional*) – Either [start, end] or list of [start, end] for the periods to be evaluated. Will be read from catalog._requested_periods if None. Leave both None to extract everything.

- **region** (*dict, optional*) – Description of the region and the subsetting method (required fields listed in the Notes) used in *xscen.spatial.subset*.
- **to_level** (*str*) – The processing level to assign to the output. Defaults to 'extracted'
- **ensure_correct_time** (*bool*) – When True (default), even if the data has the correct frequency, its time coordinate is checked so that it exactly matches the frequency code (xrfreq). For example, daily data given at noon would be transformed to be given at midnight. If the time coordinate is invalid, it raises an error.
- **xr_open_kwargs** (*dict, optional*) – A dictionary of keyword arguments to pass to *DataCatalogs.to_dataset_dict*, which will be passed to *xr.open_dataset*.
- **xr_combine_kwargs** (*dict, optional*) – A dictionary of keyword arguments to pass to *DataCatalogs.to_dataset_dict*, which will be passed to *xr.combine_by_coords*.
- **preprocess** (*callable, optional*) – If provided, call this function on each dataset prior to aggregation.
- **resample_methods** (*dict, optional*) – Dictionary where the keys are the variables and the values are the resampling method. Options for the resampling method are {'mean', 'min', 'max', 'sum', 'wind_direction'}. If the method is not given for a variable, it is guessed from the variable name and frequency, using the mapping in CVs/resampling_methods.json. If the variable is not found there, "mean" is used by default.
- **mask** (*xr.Dataset or xr.DataArray or bool*) – A mask that is applied to all variables and only keeps data where it is True. Where the mask is False, variable values are replaced by NaNs. The mask should have the same dimensions as the variables extracted. If *mask* is a dataset, the dataset should have a variable named 'mask'. If *mask* is True, it will expect a *mask* variable at xrfreq *fx* to have been extracted.

Returns

dict – Dictionary (keys = xrfreq) with datasets containing all available and computed variables, subsetted to the region, everything resampled to the requested frequency.

Notes

'region' fields:

name: str
Region name used to overwrite domain in the catalog.

method: str
['gridpoint', 'bbox', shape', 'sel']

tile_buffer: float, optional
Multiplier to apply to the model resolution.

kwargs
Arguments specific to the method used.

See also:

`intake_esm.core.esm_datastore.to_dataset_dict`, `xarray.open_dataset`, `xarray.combine_by_coords`

`xscen.extract.`**`get_warming_level`**(*realization: Dataset | DataArray | dict | Series | DataFrame | str | list*, *wl: float*, ***, *window: int = 20*, *tas_baseline_period: Sequence[str] | None = None*, *ignore_member: bool = False*, *tas_src: str | PathLike | None = None*, *return_horizon: bool = True*) → dict | list[str] | str

Use the IPCC Atlas method to return the window of time over which the requested level of global warming is first reached.

Parameters

- **realization** (*xr.Dataset, xr.DataArray, dict, str, Series or sequence of those*) – Model to be evaluated. Needs the four fields mip_era, source, experiment and member, as a dict or in a Dataset's attributes. Strings should follow this formatting: {mip_era}_{source}_{experiment}_{member}. Lists of dicts, strings or Datasets are also accepted, in which case the output will be a dict. Regex wildcards (.*) are accepted, but may lead to unexpected results. Datasets should include the catalogue attributes (starting by "cat:") required to create such a string: 'cat:mip_era', 'cat:experiment', 'cat:member', and either 'cat:source' for global models or 'cat:driving_model' for regional models. e.g. 'CMIP5_CanESM2_rcp85_r1i1p1'
- **wl** (*float*) – Warming level. e.g. 2 for a global warming level of +2 degree Celsius above the mean temperature of the *tas_baseline_period*.
- **window** (*int*) – Size of the rolling window in years over which to compute the warming level.
- **tas_baseline_period** (*list, optional*) – [start, end] of the base period. The warming is calculated with respect to it. The default is ["1850", "1900"].
- **ignore_member** (*bool*) – Decides whether to ignore the member when searching for the model run in tas_csv.
- **tas_src** (*str, optional*) – Path to a netCDF of annual global mean temperature (tas) with an annual "time" dimension and a "simulation" dimension with the following coordinates: "mip_era", "source", "experiment" and "member". If None, it will default to data/IPCC_annual_global_tas.nc which was built from the IPCC atlas data from Iturbide et al., 2020 (https://doi.org/10.5194/essd-12-2959-2020) and extra data for missing CMIP6 models and pilot models of CRCM5 and ClimEx.
- **return_horizon** (*bool*) – If True, the output will be a list following the format ['start_yr', 'end_yr'] If False, the output will be a string representing the middle of the period.

Returns

dict, list or str – If *realization* is not a sequence, the output will follow the format indicated by *return_horizon*. If *realization* is a sequence, the output will be a list or dictionary depending on *output*, with values following the format indicated by *return_horizon*.

xscen.extract.**resample**(*da: DataArray*, *target_frequency: str*, ***, *ds: Dataset | None = None*, *method: str | None = None*, *missing: str | dict | None = None*) → DataArray

Aggregate variable to the target frequency.

If the input frequency is greater than a week, the resampling operation is weighted by the number of days in each sampling period.

Parameters

- **da** (*xr.DataArray*) – DataArray of the variable to resample, must have a "time" dimension and be of a finer temporal resolution than "target_frequency".
- **target_frequency** (*str*) – The target frequency/freq str, must be one of the frequency supported by xarray.
- **ds** (*xr.Dataset, optional*) – The "wind_direction" resampling method needs extra variables, which can be given here.

- **method** (*{'mean', 'min', 'max', 'sum', 'wind_direction'}, optional*) – The resampling method. If None (default), it is guessed from the variable name and frequency, using the mapping in CVs/resampling_methods.json. If the variable is not found there, "mean" is used by default.
- **missing** (*{'mask', 'drop'} or dict, optional*) – If 'mask' or 'drop', target periods that would have been computed from fewer timesteps than expected are masked or dropped, using a threshold of 5% of missing data. E.g. the first season of a *target_frequency* of "QS-DEC" will be masked or dropped if data starts in January. If a dict, points to a xclim check missing method which will mask periods according to the number of NaN values. The dict must contain a "method" field corresponding to the xclim method name and may contain any other args to pass. Options are documented in `xclim.core.missing`.

Returns

xr.DataArray – Resampled variable

`xscen.extract.`**`search_data_catalogs`**(*data_catalogs: str | PathLike | DataCatalog | list[str | PathLike | DataCatalog]*, *variables_and_freqs: dict*, ***, *other_search_criteria: dict | None = None*, *exclusions: dict | None = None*, *match_hist_and_fut: bool = False*, *periods: list[str] | list[list[str]] | None = None*, *coverage_kwargs: dict | None = None*, *id_columns: list[str] | None = None*, *allow_resampling: bool = False*, *allow_conversion: bool = False*, *conversion_yaml: str | None = None*, *restrict_resolution: str | None = None*, *restrict_members: dict | None = None*, *restrict_warming_level: dict | bool | None = None*) → dict

Search through DataCatalogs.

Parameters

- **data_catalogs** (*str, os.PathLike, DataCatalog, or a list of those*) – DataCatalog (or multiple, in a list) or paths to JSON/CSV data catalogs. They must use the same columns and aggregation options.
- **variables_and_freqs** (*dict*) – Variables and freqs to search for, following a 'variable: xr-freq-compatible-str' format. A list of strings can also be provided.
- **other_search_criteria** (*dict, optional*) – Other criteria to search for in the catalogs' columns, following a 'column_name: list(subset)' format. You can also pass 'require_all_on: list(columns_name)' in order to only return results that correspond to all other criteria across the listed columns. More details available at https://intake-esm.readthedocs.io/en/stable/how-to/enforce-search-query-criteria-via-require-all-on.html .
- **exclusions** (*dict, optional*) – Same as other_search_criteria, but for eliminating results. Any result that matches any of the exclusions will be removed.
- **match_hist_and_fut** (*bool*) – If True, historical and future simulations will be combined into the same line, and search results lacking one of them will be rejected.
- **periods** (*list of str or list of lists of str, optional*) – Either [start, end] or list of [start, end] for the periods to be evaluated.
- **coverage_kwargs** (*dict, optional*) – Arguments to pass to subset_file_coverage (only used when periods is not None).
- **id_columns** (*list, optional*) – List of columns used to create a id column. If None is given, the original "id" is left.
- **allow_resampling** (*bool*) – If True (default), variables with a higher time resolution than requested are considered.

- **allow_conversion** (*bool*) – If True (default) and if the requested variable cannot be found, intermediate variables are searched given that there exists a converting function in the "derived variable registry".
- **conversion_yaml** (*str, optional*) – Path to a YAML file that defines the possible conversions (used alongside 'allow_conversion'=True). This file should follow the xclim conventions for building a virtual module. If None, the "derived variable registry" will be defined by the file in "xscen/xclim_modules/conversions.yml"
- **restrict_resolution** (*str, optional*) – Used to restrict the results to the finest/coarsest resolution available for a given simulation. ['finest', 'coarsest'].
- **restrict_members** (*dict, optional*) – Used to restrict the results to a given number of members for a given simulation. Currently only supports {"ordered": int} format.
- **restrict_warming_level** (*bool or dict, optional*) – Used to restrict the results only to datasets that exist in the csv used to compute warming levels in *subset_warming_level*. If True, this will only keep the datasets that have a mip_era, source, experiment and member combination that exist in the csv. This does not guarantee that a given warming level will be reached, only that the datasets have corresponding columns in the csv. More option can be added by passing a dictionary instead of a boolean. If {'ignore_member':True}, it will disregard the member when trying to match the dataset to a column. If {tas_src: Path_to_netcdf}, it will use an alternative netcdf instead of the default one provided by xscen. If 'wl' is a provided key, then *xs.get_warming_level* will be called and only datasets that reach the given warming level will be kept. This can be combined with other arguments of the function, for example {'wl': 1.5, 'window': 30}.

Notes

- The "other_search_criteria" and "exclusions" arguments accept wildcard (*) and regular expressions.
- Frequency can be wildcarded with 'NA' in the *variables_and_freqs* dict.
- Variable names cannot be wildcarded, they must be CMIP6-standard.

Returns

dict – Keys are the id and values are the DataCatalogs for each entry. A single DataCatalog can be retrieved with *concat_data_catalogs(*out.values())*. Each DataCatalog has a subset of the derived variable registry that corresponds to the needs of this specific group. Usually, each entry can be written to file in a single Dataset when using *extract_dataset* with the same arguments.

See also:

`intake_esm.core.esm_datastore.search`

`xscen.extract.`**`subset_warming_level`**(*ds: Dataset*, *wl: float | Sequence[float]*, *to_level: str = 'warminglevel-{wl}vs{period0}-{period1}'*, *wl_dim: str | bool = '+{wl}Cvs{period0}-{period1}'*, ***kwargs*) → Dataset | None

Subsets the input dataset with only the window of time over which the requested level of global warming is first reached, using the IPCC Atlas method.

Parameters

- **ds** (*xr.Dataset*) – Input dataset. The dataset should include attributes to help recognize it and find its warming levels - 'cat:mip_era', 'cat:experiment', 'cat:member', and either 'cat:source' for global models or 'cat:driving_institution' (optional) +

'cat:driving_model' for regional models. Or , it should include a *realization* dimension constructed as "{mip_era}_{source or driving_model}_{experiment}_{member}" for vectorized subsetting. Vectorized subsetting is currently only implemented for annual data.

- **wl** (*float or sequence of floats*) – Warming level. e.g. 2 for a global warming level of +2 degree Celsius above the mean temperature of the *tas_baseline_period*. Multiple levels can be passed, in which case using "{wl}" in *to_level* and *wl_dim* is not recommended. Mutliple levels are currently only implemented for annual data.
- **to_level** – The processing level to assign to the output. Use "{wl}", "{period0}" and "{period1}" in the string to dynamically include *wl*, 'tas_baseline_period[0]' and 'tas_baseline_period[1]'.
- **wl_dim** (*str or boolean, optional*) – The value to use to fill the new *warminglevel* dimension. Use "{wl}", "{period0}" and "{period1}" in the string to dynamically include *wl*, 'tas_baseline_period[0]' and 'tas_baseline_period[1]'. If None, no new dimensions will be added, invalid if *wl* is a sequence. If True, the dimension will include *wl* as numbers and units of "degC".
- ****kwargs** – Instructions on how to search for warming levels, passed to `get_warming_level()`.

Returns

xr.Dataset or None – Warming level dataset, or None if *ds* can't be subsetted for the requested warming level. The dataset will have a new dimension *warminglevel* with *wl_dim* as coordinates. If *wl* was a list or if ds had a "realization" dim, the "time" axis is replaced by a fake time starting in 1000-01-01 and with a length of *window* years. Start and end years of the subsets are bound in the new coordinate "warminglevel_bounds".

2.7.3 Regridding

Functions to regrid datasets.

`xscen.regrid.create_mask`(*ds: Dataset | DataArray*, *mask_args: dict*) → DataArray

Create a 0-1 mask based on incoming arguments.

Parameters

- **ds** (*xr.Dataset or xr.DataArray*) – Dataset or DataArray to be evaluated
- **mask_args** (*dict*) – Instructions to build the mask (required fields listed in the Notes).

Note:

'mask' fields:

variable: str, optional
Variable on which to base the mask, if ds_mask is not a DataArray.

where_operator: str, optional
Conditional operator such as '>'

where_threshold: str, optional
Value threshold to be used in conjunction with where_operator.

mask_nans: bool
Whether to apply a mask on NaNs.

Returns
xr.DataArray – Mask array.

xscen.regrid.**regrid_dataset**(*ds: Dataset*, *ds_grid: Dataset*, *weights_location: str | PathLike*, ***, *regridder_kwargs: dict | None = None*, *intermediate_grids: dict | None = None*, *to_level: str = 'regridded'*) → Dataset

Regrid a dataset according to weights and a reference grid.

Based on an intake_esm catalog, this function performs regridding on Zarr files.

Parameters

- **ds** (*xarray.Dataset*) – Dataset to regrid. The Dataset needs to have lat/lon coordinates. Supports a 'mask' variable compatible with ESMF standards.
- **weights_location** (*Union[str, os.PathLike]*) – Path to the folder where weight file is saved.
- **ds_grid** (*xr.Dataset*) – Destination grid. The Dataset needs to have lat/lon coordinates. Supports a 'mask' variable compatible with ESMF standards.
- **regridder_kwargs** (*dict, optional*) – Arguments to send xe.Regridder(). If it contains *skipna* or *out_chunks*, those are passed to the regridder call directly.
- **intermediate_grids** (*dict, optional*) – This argument is used to do a regridding in many steps, regridding to regular grids before regridding to the final ds_grid. This is useful when there is a large jump in resolution between ds and ds grid. The format is a nested dictionary shown in Notes. If None, no intermediary grid is used, there is only a regrid from ds to ds_grid.
- **to_level** (*str*) – The processing level to assign to the output. Defaults to 'regridded'

Returns
xarray.Dataset – Regridded dataset

Notes

intermediate_grids =

**{'name_of_inter_grid_1': {'cf_grid_2d': {arguments for util.cf_grid_2d
},'regridder_kwargs':{arguments for xe.Regridder}},**
'name_of_inter_grid_2': dictionary_as_above}

See also:

`xesmf.regridder`, `xesmf.util.cf_grid_2d`

2.7.4 Bias Adjustment

Functions to train and adjust a dataset using a bias-adjustment algorithm.

xscen.biasadjust.**adjust**(*dtrain: Dataset*, *dsim: Dataset*, *periods: list[str] | list[list[str]]*, ***, *xclim_adjust_args: dict | None = None*, *to_level: str = 'biasadjusted'*, *bias_adjust_institution: str | None = None*, *bias_adjust_project: str | None = None*, *moving_yearly_window: dict | None = None*, *align_on: str | None = 'year'*) → Dataset

Adjust a simulation.

Parameters

- **dtrain** (*xr.Dataset*) – A trained algorithm's dataset, as returned by *train*.

- **dsim** (*xr.Dataset*) – Simulated timeseries, projected period.
- **periods** (*list of str or list of lists of str*) – Either [start, end] or list of [start, end] of the simulation periods to be adjusted (one at a time).
- **xclim_adjust_args** (*dict, optional*) – Dict of arguments to pass to the *.adjust* of the adjustment object.
- **to_level** (*str*) – The processing level to assign to the output. Defaults to 'biasadjusted'
- **bias_adjust_institution** (*str, optional*) – The institution to assign to the output.
- **bias_adjust_project** (*str, optional*) – The project to assign to the output.
- **moving_yearly_window** (*dict, optional*) – Arguments to pass to *xclim.sdba.construct_moving_yearly_window*. If not None, *construct_moving_yearly_window* will be called on dsim (and scen in xclim_adjust_args if it exists) before adjusting and *unpack_moving_yearly_window* will be called on the output after the adjustment. *construct_moving_yearly_window* stacks windows of the dataArray in a new 'movingwin' dimension. *unpack_moving_yearly_window* unpacks it to a normal time series.
- **align_on** (*str, optional*) – *align_on* argument for the fonction *xclim.core.calendar.convert_calendar*.

Returns

xr.Dataset – dscen, the bias-adjusted timeseries.

See also:

`xclim.sdba.adjustment.DetrendedQuantileMapping`, `xclim.sdba.adjustment.ExtremeValues`

xscen.biasadjust.**train**(*dref: Dataset*, *dhist: Dataset*, *var: str | list[str]*, *period: list[str]*, ***, *method: str = 'DetrendedQuantileMapping'*, *group: Grouper | str | dict | None = None*, *xclim_train_args: dict | None = None*, *maximal_calendar: str = 'noleap'*, *adapt_freq: dict | None = None*, *jitter_under: dict | None = None*, *jitter_over: dict | None = None*, *align_on: str | None = 'year'*) → Dataset

Train a bias-adjustment.

Parameters

- **dref** (*xr.Dataset*) – The target timeseries, on the reference period.
- **dhist** (*xr.Dataset*) – The timeseries to adjust, on the reference period.
- **var** (*str or list of str*) – Variable on which to do the adjustment. Currently only supports one variable.
- **period** (*list of str*) – [start, end] of the reference period
- **method** (*str*) – Name of the *sdba.TrainAdjust* method of xclim.
- **group** (*str or sdba.Grouper or dict, optional*) – Grouping information. If a string, it is interpreted as a grouper on the time dimension. If a dict, it is passed to *sdba.Grouper.from_kwargs*. Defaults to {"group": "time.dayofyear", "window": 31}.
- **xclim_train_args** (*dict*) – Dict of arguments to pass to the *.train* of the adjustment object.
- **maximal_calendar** (*str*) – Maximal calendar dhist can be. The hierarchy: 360_day < noleap < standard < all_leap. If dhist's calendar is higher than maximal calendar, it will be converted to the maximal calendar.
- **adapt_freq** (*dict, optional*) – If given, a dictionary of args to pass to the frequency adaptation function.

- **jitter_under** (*dict, optional*) – If given, a dictionary of args to pass to *jitter_under_thresh*.
- **jitter_over** (*dict, optional*) – If given, a dictionary of args to pass to *jitter_over_thresh*.
- **align_on** (*str, optional*) – *align_on* argument for the function *xclim.core.calendar.convert_calendar*.

Returns
xr.Dataset – Trained algorithm's data.

See also:

`xclim.sdba.adjustment.DetrendedQuantileMapping`, `xclim.sdba.adjustment.ExtremeValues`

2.7.5 Indicators

Functions to compute xclim indicators.

`xscen.indicators.compute_indicators`(*ds: Dataset*, *indicators: str | PathLike | Sequence[Indicator] | Sequence[tuple[str, Indicator]] | module*, ***, *periods: list[str] | list[list[str]] | None = None*, *restrict_years: bool = True*, *to_level: str | None = 'indicators'*) → dict

Calculate variables and indicators based on a YAML call to xclim.

The function cuts the output to be the same years as the inputs. Hence, if an indicator creates a timestep outside the original year range (e.g. the first DJF for QS-DEC), it will not appear in the output.

Parameters

- **ds** (*xr.Dataset*) – Dataset to use for the indicators.
- **indicators** (*Union[str, os.PathLike, Sequence[Indicator], Sequence[tuple[str, Indicator]], ModuleType]*) – Path to a YAML file that instructs on how to calculate missing variables. Can also be only the "stem", if translations and custom indices are implemented. Can be the indicator module directly, or a sequence of indicators or a sequence of tuples (indicator name, indicator) as returned by *iter_indicators()*.
- **periods** (*list of str or list of lists of str, optional*) – Either [start, end] or list of [start, end] of continuous periods over which to compute the indicators. This is needed when the time axis of ds contains some jumps in time. If None, the dataset will be considered continuous.
- **restrict_years** (*bool*) – If True, cut the time axis to be within the same years as the input. This is mostly useful for frequencies that do not start in January, such as QS-DEC. In that instance, *xclim* would start on previous_year-12-01 (DJF), with a NaN. *restrict_years* will cut that first timestep. This should have no effect on YS and MS indicators.
- **to_level** (*str, optional*) – The processing level to assign to the output. If None, the processing level of the inputs is preserved.

Returns
dict – Dictionary (keys = timedeltas) with indicators separated by temporal resolution.

See also:

`xclim.indicators`, `xclim.core.indicator.build_indicator_module_from_yaml`

`xscen.indicators.load_xclim_module`(*filename: str | PathLike*, *reload: bool = False*) → module

Return the xclim module described by the yaml file (or group of yaml, jsons and py).

Parameters

- **filename** (*str or os.PathLike*) – The filepath to the yaml file of the module or to the stem of yaml, jsons and py files.
- **reload** (*bool*) – If False (default) and the module already exists in *xclim.indicators*, it is not re-build.

Returns
ModuleType – The xclim module.

2.7.6 Ensembles

Ensemble statistics and weights.

xscen.ensembles.**build_partition_data**(*datasets: dict | list[Dataset]*, *partition_dim: list[str] = ['source', 'experiment', 'bias_adjust_project']*, *subset_kw: dict = None*, *regrid_kw: dict = None*, *rename_dict: dict = None*)

Get the input for the xclim partition functions.

From a list or dictionary of datasets, create a single dataset with *partition_dim* dimensions (and time) to pass to one of the xclim partition functions (https://xclim.readthedocs.io/en/stable/api.html#uncertainty-partitioning). If the inputs have different grids, they have to be subsetted and regridded to a common grid/point.

Parameters

- **datasets** (*dict*) – List or dictionnary of Dataset objects that will be included in the ensemble. The datasets should include the necessary ("cat:") attributes to understand their metadata. Tip: With a project catalog, you can do: *datasets = pcat.search(**search_dict).to_dataset_dict()*.
- **partition_dim** (*list[str]*) – Components of the partition. They will become the dimension of the output. The default is ['source', 'experiment', 'bias_adjust_project']. For source, the dimension will actually be institution_source_member.
- **subset_kw** (*dict*) – Arguments to pass to *xs.spatial.subset()*.
- **regrid_kw** – Arguments to pass to *xs.regrid_dataset()*.
- **rename_dict** – Dictionary to rename the dimensions from xscen names to xclim names. The default is {'source': 'model', 'bias_adjust_project': 'downscaling', 'experiment': 'scenario'}.

Returns
xr.Dataset – The input data for the partition functions.

See also:

`xclim.ensembles`

xscen.ensembles.**ensemble_stats**(*datasets: dict | list[str | PathLike] | list[Dataset] | list[DataArray] | Dataset*, *statistics: dict*, ***, *create_kwargs: dict | None = None*, *weights: DataArray | None = None*, *common_attrs_only: bool = True*, *to_level: str = 'ensemble'*) → Dataset

Create an ensemble and computes statistics on it.

Parameters

- **datasets** (*dict or list of [str, os.PathLike, Dataset or DataArray], or Dataset*) – List of file paths or xarray Dataset/DataArray objects to include in the ensemble. A dictionary can be passed instead of a list, in which case the keys are used as coordinates along the new *realization* axis. Tip: With a project catalog, you can do: *datasets =*

*pcat.search(**search_dict).to_dataset_dict()*. If a single Dataset is passed, it is assumed to already be an ensemble and will be used as is. The 'realization' dimension is required.

- **statistics** (*dict*) – xclim.ensembles statistics to be called. Dictionary in the format {function: arguments}. If a function requires 'weights', you can leave it out of this dictionary and it will be applied automatically if the 'weights' argument is provided. See the Notes section for more details on robustness statistics, which are more complex in their usage.
- **create_kwargs** (*dict, optional*) – Dictionary of arguments for xclim.ensembles.create_ensemble.
- **weights** (*xr.DataArray, optional*) – Weights to apply along the 'realization' dimension. This array cannot contain missing values.
- **common_attrs_only** (*bool*) – If True, keeps only the global attributes that are the same for all datasets and generate new id. If False, keeps global attrs of the first dataset (same behaviour as xclim.ensembles.create_ensemble)
- **to_level** (*str*) – The processing level to assign to the output.

Returns

xr.Dataset – Dataset with ensemble statistics

Notes

- The positive fraction in 'change_significance' and 'robustness_fractions' is calculated by xclim using 'v > 0', which is not appropriate for relative deltas. This function will attempt to detect relative deltas by using the 'delta_kind' attribute ('rel.', 'relative', '*', or '/') and will apply 'v - 1' before calling the function.
- The 'robustness_categories' statistic requires the outputs of 'robustness_fractions'. Thus, there are two ways to build the 'statistics' dictionary:
  1. Having 'robustness_fractions' and 'robustness_categories' as separate entries in the dictionary. In this case, all outputs will be returned.
  2. Having 'robustness_fractions' as a nested dictionary under 'robustness_categories'. In this case, only the robustness categories will be returned.
- A 'ref' DataArray can be passed to 'change_significance' and 'robustness_fractions', which will be used by xclim to compute deltas and perform some significance tests. However, this supposes that both 'datasets' and 'ref' are still timeseries (e.g. annual means), not climatologies where the 'time' dimension represents the period over which the climatology was computed. Thus, using 'ref' is only accepted if 'robustness_fractions' (or 'robustness_categories') is the only statistic being computed.
- If you want to use compute a robustness statistic on a climatology, you should first compute the climatologies and deltas yourself, then leave 'ref' as None and pass the deltas as the 'datasets' argument. This will be compatible with other statistics.

See also:

xclim.ensembles._base.create_ensemble, xclim.ensembles._base.ensemble_percentiles, xclim.ensembles._base.ensemble_mean_std_max_min, xclim.ensembles._robustness.robustness_fractions, xclim.ensembles._robustness.robustness_categories, xclim.ensembles._robustness.robustness_coefficient

xscen.ensembles.**generate_weights**(*datasets: dict | list*, ***, *independence_level: str = 'model'*, *balance_experiments: bool = False*, *attribute_weights: dict | None = None*, *skipna: bool = True*, *v_for_skipna: str | None = None*, *standardize: bool = False*, *experiment_weights: bool = False*) → DataArray

Use realization attributes to automatically generate weights along the 'realization' dimension.

Parameters

- **datasets** (*dict*) – List of Dataset objects that will be included in the ensemble. The datasets should include the necessary attributes to understand their metadata - See 'Notes' below. A dictionary can be passed instead of a list, in which case the keys are used for the 'realization' coordinate. Tip: With a project catalog, you can do: *datasets = pcat.search(**search_dict).to_dataset_dict()*.
- **independence_level** (*str*) – 'model': Weights using the method '1 model - 1 Vote', where every unique combination of 'source' and 'driving_model' is considered a model. 'GCM': Weights using the method '1 GCM - 1 Vote' 'institution': Weights using the method '1 institution - 1 Vote'
- **balance_experiments** (*bool*) – If True, each experiment will be given a total weight of 1 (prior to subsequent weighting made through *attribute_weights*). This option requires the 'cat:experiment' attribute to be present in all datasets.
- **attribute_weights** (*dict, optional*) – Nested dictionaries of weights to apply to each dataset. These weights are applied after the independence weighting. The first level of keys are the attributes for which weights are being given. The second level of keys are unique entries for the attribute, with the value being either an individual weight or a xr.DataArray. If a DataArray is used, its dimensions must be the same non-stationary coordinate as the datasets (ex: time, horizon) and the attribute being weighted (ex: experiment). A *others* key can be used to give the same weight to all entries not specifically named in the dictionary. Example #1: {'source': {'MPI-ESM-1-2-HAM': 0.25, 'MPI-ESM1-2-HR': 0.5}}, Example #2: {'experiment': {'ssp585': xr.DataArray, 'ssp126': xr.DataArray}, 'institution': {'CCCma': 0.5, 'others': 1}}
- **skipna** (*bool*) – If True, weights will be computed from attributes only. If False, weights will be computed from the number of non-missing values. skipna=False requires either a 'time' or 'horizon' dimension in the datasets.
- **v_for_skipna** (*str, optional*) – Variable to use for skipna=False. If None, the first variable in the first dataset is used.
- **standardize** (*bool*) – If True, the weights are standardized to sum to 1 (per timestep/horizon, if skipna=False).
- **experiment_weights** (*bool*) – Deprecated. Use balance_experiments instead.

Notes

The following attributes are required for the function to work:

- 'cat:source' in all datasets
- 'cat:driving_model' in regional climate models
- 'cat:institution' in all datasets if independence_level='institution'
- 'cat:experiment' in all datasets if split_experiments=True

Even when not required, the 'cat:member' and 'cat:experiment' attributes are strongly recommended to ensure the weights are computed correctly.

Returns

xr.DataArray – Weights along the 'realization' dimension, or 2D weights along the 'realization' and 'time/horizon' dimensions if skipna=False.

2.7.7 Aggregation

Functions to aggregate data over time and space.

xscen.aggregate.**climatological_mean**(*ds: Dataset*, *, *window: int | None = None*, *min_periods: int | None = None*, *interval: int = 1*, *periods: list[str] | list[list[str]] | None = None*, *to_level: str | None = 'climatology'*) → Dataset

Compute the mean over 'year' for given time periods, respecting the temporal resolution of ds.

Parameters

- **ds** (*xr.Dataset*) – Dataset to use for the computation.
- **window** (*int, optional*) – Number of years to use for the time periods. If left at None and periods is given, window will be the size of the first period. If left at None and periods is not given, the window will be the size of the input dataset.
- **min_periods** (*int, optional*) – For the rolling operation, minimum number of years required for a value to be computed. If left at None and the xrfreq is either QS or AS and doesn't start in January, min_periods will be one less than window. If left at None, it will be deemed the same as 'window'.
- **interval** (*int*) – Interval (in years) at which to provide an output.
- **periods** (*list of str or list of lists of str, optional*) – Either [start, end] or list of [start, end] of continuous periods to be considered. This is needed when the time axis of ds contains some jumps in time. If None, the dataset will be considered continuous.
- **to_level** (*str, optional*) – The processing level to assign to the output. If None, the processing level of the inputs is preserved.

Returns

xr.Dataset – Returns a Dataset of the climatological mean, by calling climatological_op with option op=='mean'.

xscen.aggregate.**climatological_op**(*ds: Dataset*, *, *op: str | dict = 'mean'*, *window: int | None = None*, *min_periods: int | float | None = None*, *stride: int = 1*, *periods: list[str] | list[list[str]] | None = None*, *rename_variables: bool = True*, *to_level: str = 'climatology'*, *horizons_as_dim: bool = False*) → Dataset

Perform an operation 'op' over time, for given time periods, respecting the temporal resolution of ds.

Parameters

- **ds** (*xr.Dataset*) – Dataset to use for the computation.
- **op** (*str or dict*) – Operation to perform over time. The operation can be any method name of xarray.core.rolling.DatasetRolling, 'linregress', or a dictionary. If 'op' is a dictionary, the key is the operation name and the value is a dict of kwargs accepted by the operation. While other operations are technically possible, the following are recommended and tested: ['max', 'mean', 'median', 'min', 'std', 'sum', 'var', 'linregress']. Operations beyond methods of xarray.core.rolling.DatasetRolling include:
  - 'linregress' : Computes the linear regression over time, using scipy.stats.linregress and employing years as regressors. The output will have a new dimension 'linreg_param' with coordinates: ['slope', 'intercept', 'rvalue', 'pvalue', 'stderr', 'intercept_stderr'].

Only one operation per call is supported, so len(op)==1 if a dict.

- **window** (*int, optional*) – Number of years to use for the rolling operation. If left at None and periods is given, window will be the size of the first period. Hence, if periods are of different lengths, the shortest period should be passed first. If left at None and periods is not given, the window will be the size of the input dataset.
- **min_periods** (*int or float, optional*) – For the rolling operation, minimum number of years required for a value to be computed. If left at None and the xrfreq is either QS or AS and doesn't start in January, min_periods will be one less than window. Otherwise, if left at None, it will be deemed the same as 'window'. If passed as a float value between 0 and 1, this will be interpreted as the floor of the fraction of the window size.
- **stride** (*int*) – Stride (in years) at which to provide an output from the rolling window operation.
- **periods** (*list of str or list of lists of str, optional*) – Either [start, end] or list of [start, end] of continuous periods to be considered. This is needed when the time axis of ds contains some jumps in time. If None, the dataset will be considered continuous.
- **rename_variables** (*bool*) – If True, '_clim_{op}' will be added to variable names.
- **to_level** (*str, optional*) – The processing level to assign to the output. If None, the processing level of the inputs is preserved.
- **horizons_as_dim** (*bool*) – If True, the output will have 'horizon' and the frequency as 'month', 'season' or 'year' as dimensions and coordinates. The 'time' coordinate will be unstacked to horizon and frequency dimensions. Horizons originate from periods and/or windows and their stride in the rolling operation.

Returns

xr.Dataset – Dataset with the results from the climatological operation.

xscen.aggregate.**compute_deltas**(*ds: Dataset*, *reference_horizon: str | Dataset*, ***, *kind: str | dict = '+'*, *rename_variables: bool = True*, *to_level: str | None = 'deltas'*) → Dataset

Compute deltas in comparison to a reference time period, respecting the temporal resolution of ds.

Parameters

- **ds** (*xr.Dataset*) – Dataset to use for the computation.
- **reference_horizon** (*str or xr.Dataset*) – Either a YYYY-YYYY string corresponding to the 'horizon' coordinate of the reference period, or a xr.Dataset containing the climatological mean.
- **kind** (*str or dict*) – ['+', '/', '%'] Whether to provide absolute, relative, or percentage deltas. Can also be a dictionary separated per variable name.
- **rename_variables** (*bool*) – If True, '_delta_YYYY-YYYY' will be added to variable names.
- **to_level** (*str, optional*) – The processing level to assign to the output. If None, the processing level of the inputs is preserved.

Returns

xr.Dataset – Returns a Dataset with the requested deltas.

xscen.aggregate.**produce_horizon**(*ds: Dataset*, *indicators: str | PathLike | Sequence[Indicator] | Sequence[tuple[str, Indicator]] | module*, ***, *periods: list[str] | list[list[str]] | None = None*, *warminglevels: dict | None = None*, *to_level: str | None = 'horizons'*, *period: list | None = None*) → Dataset

Compute indicators, then the climatological mean, and finally unstack dates in order to have a single dataset with all indicators of different frequencies.

Once this is done, the function drops 'time' in favor of 'horizon'. This function computes the indicators and does an interannual mean. It stacks the season and month in different dimensions and adds a dimension *horizon* for the period or the warming level, if given.

Parameters

- **ds** (*xr.Dataset*) – Input dataset with a time dimension.
- **indicators** (*Union[str, os.PathLike, Sequence[Indicator], Sequence[Tuple[str, Indicator]], ModuleType]*) – Indicators to compute. It will be passed to the *indicators* argument of *xs.compute_indicators*.
- **periods** (*list of str or list of lists of str, optional*) – Either [start, end] or list of [start_year, end_year] for the period(s) to be evaluated. If both periods and warminglevels are None, the full time series will be used.
- **warminglevels** (*dict, optional*) – Dictionary of arguments to pass to *py:func:xscen.subset_warming_level*. If 'wl' is a list, the function will be called for each value and produce multiple horizons. If both periods and warminglevels are None, the full time series will be used.
- **to_level** (*str, optional*) – The processing level to assign to the output. If there is only one horizon, you can use "{wl}", "{period0}" and "{period1}" in the string to dynamically include that information in the processing level.

Returns

xr.Dataset – Horizon dataset.

xscen.aggregate.**spatial_mean**(*ds: Dataset*, *method: str*, ***, *spatial_subset: bool | None = None*, *call_clisops: bool | None = False*, *region: str | dict | None = None*, *kwargs: dict | None = None*, *simplify_tolerance: float | None = None*, *to_domain: str | None = None*, *to_level: str | None = None*) → Dataset

Compute the spatial mean using a variety of available methods.

Parameters

- **ds** (*xr.Dataset*) – Dataset to use for the computation.
- **method** (*str*) – 'cos-lat' will weight the area covered by each pixel using an approximation based on latitude. 'interp_centroid' will find the region's centroid (if coordinates are not fed through kwargs), then perform a .interp() over the spatial dimensions of the Dataset. The coordinate can also be directly fed to .interp() through the 'kwargs' argument below. 'xesmf' will make use of xESMF's SpatialAverager. This will typically be more precise, especially for irregular regions, but can be much slower than other methods.
- **spatial_subset** (*bool, optional*) – If True, xscen.spatial.subset will be called prior to the other operations. This requires the 'region' argument. If None, this will automatically become True if 'region' is provided and the subsetting method is either 'cos-lat' or 'mean'.
- **region** (*dict or str, optional*) – Description of the region and the subsetting method (required fields listed in the Notes). If method=='interp_centroid', this is used to find the region's centroid. If method=='xesmf', the bounding box or shapefile is given to SpatialAverager. Can also be "global", for global averages. This is simply a shortcut for *{'name': 'global', 'method': 'bbox', 'lon_bnds' [-180, 180], 'lat_bnds': [-90, 90]}*.

- **kwargs** (*dict, optional*) – Arguments to send to either mean(), interp() or SpatialAverager(). For SpatialAverager, one can give *skipna* or *out_chunks* here, to be passed to the averager call itself.
- **simplify_tolerance** (*float, optional*) – Precision (in degree) used to simplify a shapefile before sending it to SpatialAverager(). The simpler the polygons, the faster the averaging, but it will lose some precision.
- **to_domain** (*str, optional*) – The domain to assign to the output. If None, the domain of the inputs is preserved.
- **to_level** (*str, optional*) – The processing level to assign to the output. If None, the processing level of the inputs is preserved.

Returns

xr.Dataset – Returns a Dataset with the spatial dimensions averaged.

Notes

'region' required fields:

name: str
Region name used to overwrite domain in the catalog.

method: str
['gridpoint', 'bbox', shape', 'sel']

tile_buffer: float, optional
Multiplier to apply to the model resolution. Only used if spatial_subset==True.

kwargs
Arguments specific to the method used.

See also:

xarray.Dataset.mean, xarray.Dataset.interp, xesmf.SpatialAverager

2.7.8 Reduction

Functions to reduce an ensemble of simulations.

xscen.reduce.**build_reduction_data**(*datasets: dict | list[Dataset]*, *, *xrfreqs: list[str] | None = None*, *horizons: list[str] | None = None*) → DataArray

Construct the input required for ensemble reduction.

This will combine all variables into a single DataArray and stack all dimensions except "realization".

Parameters

- **datasets** (*Union[dict, list]*) – Dictionary of datasets in the format {"id": dataset}, or list of datasets. This can be generated by calling .to_dataset_dict() on a catalog.
- **xrfreqs** (*list of str, optional*) – List of unique frequencies across the datasets. If None, the script will attempt to guess the frequencies from the datasets' metadata or with xr.infer_freq().
- **horizons** (*list of str, optional*) – Subset of horizons on which to create the data.

Returns

xr.DataArray – 2D DataArray of dimensions "realization" and "criteria", to be used as input for ensemble reduction.

xscen.reduce.**reduce_ensemble**(*data: DataArray*, *method: str*, *kwargs: dict*)

Reduce an ensemble of simulations using clustering algorithms from xclim.ensembles.

Parameters

- **data** (*xr.DataArray*) – Selection criteria data : 2-D xr.DataArray with dimensions 'realization' and 'criteria'. These are the values used for clustering. Realizations represent the individual original ensemble members and criteria the variables/indicators used in the grouping algorithm. This data can be generated using build_reduction_data().
- **method** (*str*) – ['kkz', 'kmeans']. Clustering method.
- **kwargs** (*dict*) – Arguments to send to either xclim.ensembles.kkz_reduce_ensemble or xclim.ensembles.kmeans_reduce_ensemble

Returns

- **selected** (*xr.DataArray*) – DataArray of dimension 'realization' with the selected simulations.
- **clusters** (*dict*) – If using kmeans clustering, realizations grouped by cluster.
- **fig_data** (*dict*) – If using kmeans clustering, data necessary to call xclim.ensembles.plot_rsqprofile()

2.7.9 Diagnostics and Quality Checks

Functions to perform diagnostics on datasets.

xscen.diagnostics.**health_checks**(*ds: Dataset | DataArray*, ***, *structure: dict | None = None*, *calendar: str | None = None*, *start_date: str | None = None*, *end_date: str | None = None*, *variables_and_units: dict | None = None*, *cfchecks: dict | None = None*, *freq: str | None = None*, *missing: dict | str | list | None = None*, *flags: dict | None = None*, *flags_kwargs: dict | None = None*, *return_flags: bool = False*, *raise_on: list | None = None*) → None | Dataset

Perform a series of health checks on the dataset. Be aware that missing data checks and flag checks can be slow.

Parameters

- **ds** (*xr.Dataset or xr.DataArray*) – Dataset to check.
- **structure** (*dict, optional*) – Dictionary with keys "dims" and "coords" containing the expected dimensions and coordinates. This check will fail is extra dimensions or coordinates are found.
- **calendar** (*str, optional*) – Expected calendar. Synonyms should be detected correctly (e.g. "standard" and "gregorian").
- **start_date** (*str, optional*) – To check if the dataset starts at least at this date.
- **end_date** (*str, optional*) – To check if the dataset ends at least at this date.
- **variables_and_units** (*dict, optional*) – Dictionary containing the expected variables and units.

- **cfchecks** (*dict, optional*) – Dictionary where the key is the variable to check and the values are the cfchecks. The cfchecks themselves must be a dictionary with the keys being the cfcheck names and the values being the arguments to pass to the cfcheck. See *xclim.core.cfchecks* for more details.
- **freq** (*str, optional*) – Expected frequency, written as the result of xr.infer_freq(ds.time).
- **missing** (*dict or str or list of str, optional*) – String, list of strings, or dictionary where the key is the method to check for missing data and the values are the arguments to pass to the method. The methods are: "missing_any", "at_least_n_valid", "missing_pct", "missing_wmo". See `xclim.core.missing()` for more details.
- **flags** (*dict, optional*) – Dictionary where the key is the variable to check and the values are the flags. The flags themselves must be a dictionary with the keys being the data_flags names and the values being the arguments to pass to the data_flags. If *None* is passed instead of a dictionary, then xclim's default flags for the given variable are run. See `xclim.core.utils.VARIABLES`. See also `xclim.core.dataflags.data_flags()` for the list of possible flags.
- **flags_kwargs** (*dict, optional*) – Additional keyword arguments to pass to the data_flags ("dims" and "freq").
- **return_flags** (*bool*) – Whether to return the Dataset created by data_flags.
- **raise_on** (*list of str, optional*) – Whether to raise an error if a check fails, else there will only be a warning. The possible values are the names of the checks. Use ["all"] to raise on all checks.

Returns

xr.Dataset or None – Dataset containing the flags if return_flags is True & raise_on is False for the "flags" check.

xscen.diagnostics.**measures_heatmap**(*meas_datasets: list[Dataset] | dict*, *to_level: str = 'diag-heatmap'*) → Dataset

Create a heatmap to compare the performance of the different datasets.

The columns are properties and the rows are datasets. Each point is the absolute value of the mean of the measure over the whole domain. Each column is normalized from 0 (best) to 1 (worst).

Parameters

- **meas_datasets** (*list of xr.Dataset or dict*) – List or dictionary of datasets of measures of properties. If it is a dictionary, the keys will be used to name the rows. If it is a list, the rows will be given a number.
- **to_level** (*str*) – The processing_level to assign to the output.

Returns

xr.Dataset – Dataset containing the heatmap.

xscen.diagnostics.**measures_improvement**(*meas_datasets: list[Dataset] | dict*, *to_level: str = 'diag-improved'*) → Dataset

Calculate the fraction of improved grid points for each property between two datasets of measures.

Parameters

- **meas_datasets** (*list of xr.Dataset or dict*) – List of 2 datasets: Initial dataset of measures and final (improved) dataset of measures. Both datasets must have the same variables. It is also possible to pass a dictionary where the values are the datasets and the key are not used.
- **to_level** (*str*) – processing_level to assign to the output dataset

Returns

xr.Dataset – Dataset containing information on the fraction of improved grid points for each property.

xscen.diagnostics.**properties_and_measures**(*ds: Dataset*, *properties: str | PathLike | Sequence[Indicator] | Sequence[tuple[str, Indicator]] | module*, *period: list[str] | None = None*, *unstack: bool = False*, *rechunk: dict | None = None*, *dref_for_measure: Dataset | None = None*, *change_units_arg: dict | None = None*, *to_level_prop: str = 'diag-properties'*, *to_level_meas: str = 'diag-measures'*) → tuple[Dataset, Dataset]

Calculate properties and measures of a dataset.

Parameters

- **ds** (*xr.Dataset*) – Input dataset.
- **properties** (*Union[str, os.PathLike, Sequence[Indicator], Sequence[tuple[str, Indicator]], ModuleType]*) – Path to a YAML file that instructs on how to calculate properties. Can be the indicator module directly, or a sequence of indicators or a sequence of tuples (indicator name, indicator) as returned by *iter_indicators()*.
- **period** (*list of str, optional*) – [start, end] of the period to be evaluated. The period will be selected on ds and dref_for_measure if it is given.
- **unstack** (*bool*) – Whether to unstack ds before computing the properties.
- **rechunk** (*dict, optional*) – Dictionary of chunks to use for a rechunk before computing the properties.
- **dref_for_measure** (*xr.Dataset, optional*) – Dataset of properties to be used as the ref argument in the computation of the measure. Ideally, this is the first output (prop) of a previous call to this function. Only measures on properties that are provided both in this dataset and in the properties list will be computed. If None, the second output of the function (meas) will be an empty Dataset.
- **change_units_arg** (*dict, optional*) – If not None, calls *xscen.utils.change_units* on ds before computing properties using this dictionary for the *variables_and_units* argument. It can be useful to convert units before computing the properties, because it is sometimes easier to convert the units of the variables than the units of the properties (e.g. variance).
- **to_level_prop** (*str*) – processing_level to give the first output (prop)
- **to_level_meas** (*str*) – processing_level to give the second output (meas)

Returns

- **prop** (*xr.Dataset*) – Dataset of properties of ds
- **meas** (*xr.Dataset*) – Dataset of measures between prop and dref_for_meas

See also:

`xclim.sdba.properties`, `xclim.sdba.measures`, `xclim.core.indicator.build_indicator_module_from_yaml`

2.7.10 Input / Output

Input/Output functions for xscen.

xscen.io.**clean_incomplete**(*path: str | PathLike*, *complete: Sequence[str]*) → None

Delete un-catalogued variables from a zarr folder.

The goal of this function is to clean up an incomplete calculation. It will remove any variable in the zarr that is neither in the *complete* list nor in the *coords*.

Parameters

- **path** (*str, Path*) – A path to a zarr folder.
- **complete** (*sequence of strings*) – Name of variables that were completed.

Returns

None

xscen.io.**estimate_chunks**(*ds: str | PathLike | Dataset*, *dims: list*, *target_mb: float = 50*, *chunk_per_variable: bool = False*) → dict

Return an approximate chunking for a file or dataset.

Parameters

- **ds** (*xr.Dataset, str*) – Either a xr.Dataset or the path to a NetCDF file. Existing chunks are not taken into account.
- **dims** (*list*) – Dimension(s) on which to estimate the chunking. Not implemented for more than 2 dimensions.
- **target_mb** (*float*) – Roughly the size of chunks (in Mb) to aim for.
- **chunk_per_variable** (*bool*) – If True, the output will be separated per variable. Otherwise, a common chunking will be found.

Returns

dict – A dictionary mapping dimensions to chunk sizes.

xscen.io.**get_engine**(*file: str | PathLike*) → str

Use functionality of h5py to determine if a NetCDF file is compatible with h5netcdf.

Parameters

file (*str or os.PathLike*) – Path to the file.

Returns

str – Engine to use with xarray

xscen.io.**make_toc**(*ds: Dataset | DataArray*, *loc: str | None = None*) → DataFrame

Make a table of content describing a dataset's variables.

This return a simple DataFrame with variable names as index, the long_name as "description" and units. Column names and long names are taken from the activated locale if found, otherwise the english version is taken.

Parameters

- **ds** (*xr.Dataset or xr.DataArray*) – Dataset or DataArray from which to extract the relevant metadata.
- **loc** (*str, optional*) – The locale to use. If None, either the first locale in the list of activated xclim locales is used, or "en" if none is activated.

Returns

pd.DataFrame – A DataFrame with variables as index, and columns "description" and "units".

xscen.io.**rechunk**(*path_in: PathLike | str | Dataset*, *path_out: PathLike | str*, ***, *chunks_over_var: dict | None = None*, *chunks_over_dim: dict | None = None*, *worker_mem: str*, *temp_store: str | PathLike | None = None*, *overwrite: bool = False*) → None

Rechunk a dataset into a new zarr.

Parameters

- **path_in** (*path, str or xr.Dataset*) – Input to rechunk.
- **path_out** (*path or str*) – Path to the target zarr.
- **chunks_over_var** (*dict*) – Mapping from variables to mappings from dimension name to size. Give this argument or *chunks_over_dim*.
- **chunks_over_dim** (*dict*) – Mapping from dimension name to size that will be used for all variables in ds. Give this argument or *chunks_over_var*.
- **worker_mem** (*str*) – The maximal memory usage of each task. When using a distributed Client, this an approximate memory per thread. Each worker of the client should have access to 10-20% more memory than this times the number of threads.
- **temp_store** (*path or str, optional*) – A path to a zarr where to store intermediate results.
- **overwrite** (*bool*) – If True, it will delete whatever is in path_out before doing the rechunking.

Returns

None

See also:

rechunker.rechunk

xscen.io.**rechunk_for_saving**(*ds: Dataset*, *rechunk: dict*)

Rechunk before saving to .zarr or .nc, generalized as Y/X for different axes lat/lon, rlat/rlon.

Parameters

- **ds** (*xr.Dataset*) – The xr.Dataset to be rechunked.
- **rechunk** (*dict*) – A dictionary with the dimension names of ds and the new chunk size. Spatial dimensions can be provided as X/Y.

Returns

xr.Dataset – The dataset with new chunking.

xscen.io.**round_bits**(*da: DataArray*, *keepbits: int*)

Round floating point variable by keeping a given number of bits in the mantissa, dropping the rest. This allows for a much better compression.

Parameters

- **da** (*xr.DataArray*) – Variable to be rounded.
- **keepbits** (*int*) – The number of bits of the mantissa to keep.

xscen.io.**save_to_netcdf**(*ds: Dataset*, *filename: str | PathLike*, ***, *rechunk: dict | None = None*, *bitround: bool | int | dict = False*, *compute: bool = True*, *netcdf_kwargs: dict | None = None*)

Save a Dataset to NetCDF, rechunking or compressing if requested.

Parameters

- **ds** (*xr.Dataset*) – Dataset to be saved.
- **filename** (*str or os.PathLike*) – Name of the NetCDF file to be saved.

- **rechunk** (*dict, optional*) – This is a mapping from dimension name to new chunks (in any format understood by dask). Spatial dimensions can be generalized as 'X' and 'Y', which will be mapped to the actual grid type's dimension names. Rechunking is only done on *data* variables sharing dimensions with this argument.
- **bitround** (*bool or int or dict*) – If not False, float variables are bit-rounded by dropping a certain number of bits from their mantissa, allowing for a much better compression. If an int, this is the number of bits to keep for all float variables. If a dict, a mapping from variable name to the number of bits to keep. If True, the number of bits to keep is guessed based on the variable's name, defaulting to 12, which yields a relative error below 0.013%.
- **compute** (*bool*) – Whether to start the computation or return a delayed object.
- **netcdf_kwargs** (*dict, optional*) – Additional arguments to send to_netcdf()

Returns

None

See also:

`xarray.Dataset.to_netcdf`

`xscen.io.save_to_table`(*ds: Dataset | DataArray, filename: str | PathLike, output_format: str | None = None, *, row: str | Sequence[str] | None = None, column: None | str | Sequence[str] = 'variable', sheet: str | Sequence[str] | None = None, coords: bool | Sequence[str] = True, col_sep: str = '_', row_sep: str | None = None, add_toc: bool | DataFrame = False, **kwargs*)

Save the dataset to a tabular file (csv, excel, ...).

This function will trigger a computation of the dataset.

Parameters

- **ds** (*xr.Dataset or xr.DataArray*) – Dataset or DataArray to be saved. If a Dataset with more than one variable is given, the dimension "variable" must appear in one of *row*, *column* or *sheet*.
- **filename** (*str or os.PathLike*) – Name of the file to be saved.
- **output_format** (*{'csv', 'excel', ...}, optional*) – The output format. If None (default), it is inferred from the extension of *filename*. Not all possible output format are supported for inference. Valid values are any that matches a `pandas.DataFrame` method like "df.to_{format}".
- **row** (*str or sequence of str, optional*) – Name of the dimension(s) to use as indexes (rows). Default is all data dimensions.
- **column** (*str or sequence of str, optional*) – Name of the dimension(s) to use as columns. Default is "variable", i.e. the name of the variable(s).
- **sheet** (*str or sequence of str, optional*) – Name of the dimension(s) to use as sheet names. Only valid if the output format is excel.
- **coords** (*bool or sequence of str*) – A list of auxiliary coordinates to add to the columns (as would variables). If True, all (if any) are added.
- **col_sep** (*str,*) – Multi-columns (except in excel) and sheet names are concatenated with this separator.
- **row_sep** (*str, optional*) – Multi-index names are concatenated with this separator, except in excel. If None (default), each level is written in its own column.
- **add_toc** (*bool or DataFrame*) – A table of content to add as the first sheet. Only valid if the output format is excel. If True, *make_toc()* is used to generate the toc. The sheet

name of the toc can be given through the "name" attribute of the DataFrame, otherwise "Content" is used.

- **kwargs** – Other arguments passed to the pandas function. If the output format is excel, kwargs to `pandas.ExcelWriter` can be given here as well.

`xscen.io.save_to_zarr`(*ds: Dataset*, *filename: str | PathLike*, ***, *rechunk: dict | None = None*, *zarr_kwargs: dict | None = None*, *compute: bool = True*, *encoding: dict | None = None*, *bitround: bool | int | dict = False*, *mode: str = 'f'*, *itervar: bool = False*, *timeout_cleanup: bool = True*)

Save a Dataset to Zarr format, rechunking and compressing if requested.

According to mode, removes variables that we don't want to re-compute in ds.

Parameters

- **ds** (*xr.Dataset*) – Dataset to be saved.
- **filename** (*str*) – Name of the Zarr file to be saved.
- **rechunk** (*dict, optional*) – This is a mapping from dimension name to new chunks (in any format understood by dask). Spatial dimensions can be generalized as 'X' and 'Y' which will be mapped to the actual grid type's dimension names. Rechunking is only done on *data* variables sharing dimensions with this argument.
- **zarr_kwargs** (*dict, optional*) – Additional arguments to send to_zarr()
- **compute** (*bool*) – Whether to start the computation or return a delayed object.
- **mode** (*{'f', 'o', 'a'}*) – If 'f', fails if any variable already exists. if 'o', removes the existing variables. if 'a', skip existing variables, writes the others.
- **encoding** (*dict, optional*) – If given, skipped variables are popped in place.
- **bitround** (*bool or int or dict*) – If not False, float variables are bit-rounded by dropping a certain number of bits from their mantissa, allowing for a much better compression. If an int, this is the number of bits to keep for all float variables. If a dict, a mapping from variable name to the number of bits to keep. If True, the number of bits to keep is guessed based on the variable's name, defaulting to 12, which yields a relative error of 0.012%.
- **itervar** (*bool*) – If True, (data) variables are written one at a time, appending to the zarr. If False, this function computes, no matter what was passed to kwargs.
- **timeout_cleanup** (*bool*) – If True (default) and a `xscen.scripting.TimeoutException` is raised during the writing, the variable being written is removed from the dataset as it is incomplete. This does nothing if *compute* is False.

Returns

dask.delayed object if compute=False, None otherwise.

See also:

`xarray.Dataset.to_zarr`

`xscen.io.subset_maxsize`(*ds: Dataset*, *maxsize_gb: float*) → list

Estimate a dataset's size and, if higher than the given limit, subset it alongside the 'time' dimension.

Parameters

- **ds** (*xr.Dataset*) – Dataset to be saved.
- **maxsize_gb** (*float*) – Target size for the NetCDF files. If the dataset is bigger than this number, it will be separated alongside the 'time' dimension.

Returns

list – List of xr.Dataset subsetted alongside 'time' to limit the filesize to the requested maximum.

xscen.io.**to_table**(*ds: Dataset | DataArray*, ***, *row: str | Sequence[str] | None = None*, *column: str | Sequence[str] | None = None*, *sheet: str | Sequence[str] | None = None*, *coords: bool | str | Sequence[str] = True*) → DataFrame | dict

Convert a dataset to a pandas DataFrame with support for multicolumns and multisheet.

This function will trigger a computation of the dataset.

Parameters

- **ds** (*xr.Dataset or xr.DataArray*) – Dataset or DataArray to be saved. If a Dataset with more than one variable is given, the dimension "variable" must appear in one of *row*, *column* or *sheet*.
- **row** (*str or sequence of str, optional*) – Name of the dimension(s) to use as indexes (rows). Default is all data dimensions.
- **column** (*str or sequence of str, optional*) – Name of the dimension(s) to use as columns. Default is "variable", i.e. the name of the variable(s).
- **sheet** (*str or sequence of str, optional*) – Name of the dimension(s) to use as sheet names.
- **coords** (*bool or str or sequence of str*) – A list of auxiliary coordinates to add to the columns (as would variables). If True, all (if any) are added.

Returns

pd.DataFrame or dict – DataFrame with a MultiIndex with levels *row* and MultiColumn with levels *column*. If *sheet* is given, the output is dictionary with keys for each unique "sheet" dimensions tuple, values are DataFrames. The DataFrames are always sorted with level priority as given in *row* and in ascending order.

2.7.11 Spatial tools

Spatial tools.

xscen.spatial.**creep_fill**(*da: DataArray*, *w: DataArray*) → DataArray

Creep fill using pre-computed weights.

Parameters

- **da** (*DataArray*) – A DataArray sharing the dimensions with the one used to compute the weights. It can have other dimensions. Dask is supported as long as there are no chunks over the creeped dims.
- **w** (*DataArray*) – The result of *creep_weights*.

Returns

xarray.DataArray, same shape as *da*, but values filled according to *w*.

Examples

```
>>> w = creep_weights(da.isel(time=0).notnull(), n=1)
>>> da_filled = creep_fill(da, w)
```

xscen.spatial.**creep_weights**(*mask: DataArray*, *n: int = 1*, *mode: str = 'clip'*) → DataArray

Compute weights for the creep fill.

The output is a sparse matrix with the same dimensions as *mask*, twice.

Parameters

- **mask** (*DataArray*) – A boolean DataArray. False values are candidates to the filling. Usually they represent missing values (*mask = da.notnull()*). All dimensions are creep filled.
- **n** (*int*) – The order of neighbouring to use. 1 means only the adjacent grid cells are used.
- **mode** (*{'clip', 'wrap'}*) – If a cell is on the edge of the domain, *mode='wrap'* will wrap around to find neighbours.

Returns

DataArray – Weights. The dot product must be taken over the last N dimensions.

xscen.spatial.**subset**(*ds: Dataset*, *region: dict | None = None*, ***, *name: str | None = None*, *method: str | None = None*, *tile_buffer: float = 0*, ***kwargs*) → Dataset

Subset the data to a region.

Either creates a slice and uses the .sel() method, or customizes a call to clisops.subset() that allows for an automatic buffer around the region.

Parameters

- **ds** (*xr.Dataset*) – Dataset to be subsetted.
- **region** (*dict*) – Deprecated argument that is there for legacy reasons and will be abandoned eventually.
- **name** (*str, optional*) – Used to rename the 'cat:domain' attribute.
- **method** (*str*) – ['gridpoint', 'bbox', shape','sel'] If the method is *sel*, this is not a call to clisops but only a subsetting with the xarray .sel() fonction.
- **tile_buffer** (*float*) – For ['bbox', shape'], uses an approximation of the grid cell size to add a buffer around the requested region. This differs from clisops' 'buffer' argument in subset_shape().
- **kwargs** (*dict*) – Arguments to be sent to clisops. If the method is *sel*, the keys are the dimensions to subset and the values are turned into a slice.

Returns

xr.Dataset – Subsetted Dataset.

See also:

`clisops.core.subset.subset_gridpoint`, `clisops.core.subset.subset_bbox`, `clisops.core.subset.subset_shape`

2.7.12 Controlled Vocabulary and Mappings

Mappings of (controlled) vocabulary. This module is generated automatically from json files in xscen/CVs. Functions are essentially mappings, most of which are meant to provide translations between columns.

Json files must be shallow dictionaries to be supported. If the json file contains a `is_regex: True` entry, then the keys are automatically translated as regex patterns and the function returns the value of the first key that matches the pattern. Otherwise the function essentially acts like a normal dictionary. The 'raw' data parsed from the json file is added in the `dict` attribute of the function. Example:

```
xs.utils.CV.frequency_to_timedelta.dict
```

Listing 1: frequency_to_timedelta

```
{
  "1hr": "1H",
  "3hr": "3H",
  "6hr": "6H",
  "day": "1D",
  "sem": "1W",
  "2sem": "2W",
  "mon": "30D",
  "qtr": "90D",
  "6mon": "180D",
  "yr": "365D",
  "fx": "NAN"
}
```

Listing 2: frequency_to_xrfreq

```
{
  "1hr": "H",
  "3hr": "3H",
  "6hr": "6H",
  "day": "D",
  "sem": "W",
  "2sem": "2W",
  "mon": "MS",
  "qtr": "QS-DEC",
  "6mon": "2QS-DEC",
  "yr": "YS",
  "fx": "fx"
}
```

Listing 3: infer_resolution

```
{
  "CMIP": [
    "^gn[a-g]{0,1}$",
    "^gr[0-9]{0,1}[a-g]{0,1}$",
    "^global$",
    "^gnz$",
    "^gr[0-9]{1}z$",
```

(continues on next page)

(continued from previous page)

```
    "^gm"
  ],
  "CORDEX": [
    "^[A-Z]{3}-[0-9]{2}[i]{0,1}$",
    "^[A-Z]{3}-[0-9]{2}i$"
  ]
}
```

Listing 4: resampling_methods

```
{
  "any": {
    "sfcWindfromdir": "wind_direction",
    "sfcWind": "wind_direction",
    "uas": "wind_direction",
    "vas": "wind_direction"
  },
  "D": {
    "tasmin": "min",
    "tasmax": "max"
  }
}
```

Listing 5: variable_names

```
{
  "latitude": "lat",
  "longitude": "lon",
  "t2m": "tas",
  "d2m": "tdps",
  "tp": "pr",
  "u10": "uas",
  "v10": "vas"
}
```

Listing 6: xrfreq_to_frequency

```
{
  "is_regex": true,
  "H": "1hr",
  "3H": "3hr",
  "6H": "6hr",
  "D": "day",
  "W": "sem",
  "2W": "2sem",
  "14d": "2sem",
  "M.*": "mon",
  "Q.*": "qtr",
  "2Q.*": "6mon",
  "A.*": "yr",
  "YS": "yr",
  "fx": "fx"
```

(continues on next page)

(continued from previous page)

```
}
```

Listing 7: xrfreq_to_timedelta

```
{
  "is_regex": true,
  "H": "1H",
  "3H": "3H",
  "6H": "6H",
  "D": "1D",
  "W": "7D",
  "2W": "14D",
  "M.*": "30D",
  "Q.*": "90D",
  "2Q.*": "180D",
  "A.*": "365D",
  "YS": "365D",
  "fx": "NAN"
}
```

2.7.13 Configuration Utilities

Configuration module.

Configuration in this module is taken from yaml files.

Functions wrapped by *parse_config()* have their kwargs automatically patched by values in the config.

The `CONFIG` dictionary contains all values, structured by submodules and functions. For example, for function `function` defined in `module.py` of this package, the config would look like:

```
module:
    function:
        ...kwargs...
```

The *load_config()* function fills the `CONFIG` dict from yaml files. It always updates the dictionary, so the latest file read has the highest priority.

At calling time, the priority order is always (from highest to lowest priority):

1. Explicitly passed keyword-args
2. Values in the loaded config
3. Function's default values.

Special sections

After parsing the files, *load_config()* will look into the config and perform some extra actions when finding the following special sections:

- `logging`: The content of this section will be sent directly to `logging.config.dictConfig()`.
- `xarray`: The content of this section will be sent directly to `xarray.set_options()`.
- `xclim`: The content of this section will be sent directly to `xclim.set_options()`. Here goes *metadata_locales: - fr* to activate the automatic translation of added attributes, for example.
- `warnings`: The content of this section must be a simple mapping. The keys are understood as python warning categories (types) and the values as an action to add to the filter. The key "all" applies the filter to any warnings. Only built-in warnings are supported.

xscen.config.**args_as_str**(**args: tuple[Any, ...]*) → tuple[str, ...]

Return arguments as strings.

xscen.config.**load_config**(**elements*, *reset: bool = False*, *verbose: bool = False*)

Load configuration from given files or key=value pairs.

Once all elements are loaded, special sections are dispatched to their module, but only if the section was changed by the loaded elements. These special sections are:

- *locales* : The locales to use when writing metadata in xscen, xclim and figanos. This section must be a list of 2-char strings.
- *logging* : Everything passed to `logging.config.dictConfig()`.
- *xarray* : Passed to `xarray.set_options()`.
- *xclim* : Passed to `xclim.set_options()`.
- *warning* : Mappings where the key is a Warning category (or "all") and the value an action to pass to `warnings.simplefilter()`.

Parameters

- **elements** (*str*) – Files or values to add into the config. If a directory is passed, all *.yml* files of this directory are added, in alphabetical order. If a "key=value" string, "key" is a dotted name and value will be evaluated if possible. "key=value" pairs are set last, after all files are being processed.
- **reset** (*bool*) – If True, the current config is erased before loading files.
- **verbose** (*bool*) – if True, each element triggers a INFO log line.

Example

```
load_config("my_config.yml", "config_dir/", "logging.loggers.xscen.level=DEBUG")
```

Will load configuration from *my_config.yml*, then from all yml files in *config_dir* and then the logging level of xscen's logger will be set to DEBUG.

xscen.config.**parse_config**(*func_or_cls*)

xscen.config.**recursive_update**(*d*, *other*)

Update a dictionary recursively with another dictionary.

Values that are Mappings are updated recursively as well.

2.7.14 Script Utilities

A collection of various convenience objects and functions to use in scripts.

exception xscen.scripting.**TimeoutException**(*seconds: int*, *task: str = ''*, ***kwargs*)

An exception raised with a timeout occurs.

class xscen.scripting.**measure_time**(*name: str | None = None*, *cpu: bool = False*, *logger: ~logging.Logger = <Logger xscen.scripting (INFO)>*)

Context for timing a code block.

Parameters

- **name** (*str, optional*) – A name to give to the block being timed, for meaningful logging.
- **cpu** (*boolean*) – If True, the CPU time is also measured and logged.
- **logger** (*logging.Logger, optional*) – The logger object to use when sending Info messages with the measured time. Defaults to a logger from this module.

xscen.scripting.**move_and_delete**(*moving: list[list[str | PathLike]]*, *pcat:* ProjectCatalog, *deleting: list[str | PathLike] | None = None*, *copy: bool = False*)

First, move files, then update the catalog with new locations. Finally, delete directories.

This function can be used at the end of for loop in a workflow to clean temporary files.

Parameters

- **moving** (*list of lists of str or os.PathLike*) – list of lists of path of files to move, following the format: [[source 1, destination1], [source 2, destination2],...]
- **pcat** (*ProjectCatalog*) – Catalog to update with new destinations
- **deleting** (*list of str or os.PathLike, optional*) – list of directories to be deleted including all contents and recreated empty. E.g. the working directory of a workflow.
- **copy** (*bool, optional*) – If True, copy directories instead of moving them.

xscen.scripting.**save_and_update**(*ds:* Dataset, *pcat:* ProjectCatalog, *path: str | PathLike | None = None*, *file_format: str | None = None*, *build_path_kwargs: dict | None = None*, *save_kwargs: dict | None = None*, *update_kwargs: dict | None = None*)

Construct the path, save and delete.

This function can be used after each task of a workflow.

Parameters

- **ds** (*xr.Dataset*) – Dataset to save.
- **pcat** (*ProjectCatalog*) – Catalog to update after saving the dataset.
- **path** (*str or os.pathlike, optional*) – Path where to save the dataset. If the string contains variables in curly bracket. They will be filled by catalog attributes. If None, the *catutils.build_path* fonction will be used to create a path.
- **file_format** (*{'nc', 'zarr'}*) – Format of the file. If None, look for the following in order: build_path_kwargs['format'], a suffix in path, ds.attrs['cat:format']. If nothing is found, it will default to zarr.
- **build_path_kwargs** (*dict, optional*) – Arguments to pass to *build_path*.
- **save_kwargs** (*dict, optional*) – Arguments to pass to *save_to_netcdf* or *save_to_zarr*.
- **update_kwargs** (*dict, optional*) – Arguments to pass to *update_from_ds*.

xscen.scripting.**send_mail**(*, *subject: str*, *msg: str*, *to: str | None = None*, *server: str = '127.0.0.1'*, *port: int = 25*, *attachments: list[tuple[str, Figure | PathLike] | Figure | PathLike] | None = None*) → None

Send email.

Email a single address through a login-less SMTP server. The default values of server and port should work out-of-the-box on Ouranos's systems.

Parameters

- **subject** (*str*) – Subject line.
- **msg** (*str*) – Main content of the email. Can be UTF-8 and multi-line.
- **to** (*str, optional*) – Email address to which send the email. If None (default), the email is sent to "{os.getlogin()}@{os.uname().nodename}". On unix systems simply put your real email address in *$HOME/.forward* to receive the emails sent to this local address.
- **server** (*str*) – SMTP server url. Defaults to 127.0.0.1, the local host. This function does not try to log-in.
- **port** (*int*) – Port of the SMTP service on the server. Defaults to 25, which is usually the default port on unix-like systems.
- **attachments** (*list of paths or matplotlib figures or tuples of a string and a path or figure, optional*) – List of files to attach to the email. Elements of the list can be paths, the mime-types of those is guessed and the files are read and sent. Elements can also be matplotlib Figures which are send as png image (savefig) with names like "Figure00.png". Finally, elements can be tuples of a filename to use in the email and the attachment, handled as above.

Returns

None

xscen.scripting.**send_mail_on_exit**(*, *subject: str | None = None*, *msg_ok: str | None = None*, *msg_err: str | None = None*, *on_error_only: bool = False*, *skip_ctrlc: bool = True*, ***mail_kwargs*) → None

Send an email with content depending on how the system exited.

This function is best used by registering it with *atexit*. Calls `send_mail()`.

Parameters

- **subject** (*str, optional*) – Email subject. Will be appended by "Success", "No errors" or "Failure" depending on how the system exits.
- **msg_ok** (*str, optional*) – Content of the email if the system exists successfully.
- **msg_err** (*str, optional*) – Content of the email id the system exists with a non-zero code or with an error. The message will be appended by the exit code or with the error traceback.
- **on_error_only** (*boolean*) – Whether to only send an email on a non-zero/error exit.
- **skip_ctrlc** (*boolean*) – If True (default), exiting with a KeyboardInterrupt will not send an email.
- **mail_kwargs** – Other arguments passed to `send_mail()`. The *to* argument is necessary for this function to work.

Returns

None

Example

Send an eamil titled "Woups" upon non-successful program exit. We assume the *to* field was given in the config.

```
>>> import atexit
>>> atexit.register(send_mail_on_exit, subject="Woups", on_error_only=True)
```

xscen.scripting.**skippable**(*seconds: int = 2*, *task: str = ''*, *logger: Logger | None = None*)

Skippable context manager.

When CTRL-C (SIGINT, KeyboardInterrupt) is sent within the context, this catches it, prints to the log and gives a timeout during which a subsequent interruption will stop the script. Otherwise, the context exits normally.

This is meant to be used within a loop so that we can skip some iterations:

```
for i in iterable:
    with skippable(2, i):
        some_skippable_code()
```

Parameters

- **seconds** (*int*) – Number of seconds to wait for a second CTRL-C.
- **task** (*str*) – A name for the skippable task, to have an explicit script.
- **logger** (*logging.Logger, optional*) – The logger to use when printing the messages. The interruption signal is notified with ERROR, while the skipping is notified with INFO. If not given (default), a brutal print is used.

xscen.scripting.**timeout**(*seconds: int*, *task: str = ''*)

Timeout context manager.

Only one can be used at a time, this is not multithread-safe : it cannot be used in another thread than the main one, but multithreading can be used in parallel.

Parameters

- **seconds** (*int*) – Number of seconds after which the context exits with a TimeoutException. If None or negative, no timeout is set and this context does nothing.
- **task** (*str, optional*) – A name to give to the task, allowing a more meaningful exception.

2.7.15 Packaging Utilities

Common utilities to be used in many places.

xscen.utils.**add_attr**(*ds: Dataset | DataArray*, *attr: str*, *new: str*, ***fmt*)

Add a formatted translatable attribute to a dataset.

xscen.utils.**change_units**(*ds: Dataset*, *variables_and_units: dict*) → Dataset

Change units of Datasets to non-CF units.

Parameters

- **ds** (*xr.Dataset*) – Dataset to use
- **variables_and_units** (*dict*) – Description of the variables and units to output

Returns

xr.Dataset

See also:

xclim.core.units.convert_units_to, xclim.core.units.rate2amount

xscen.utils.**clean_up**(*ds: Dataset*, ***, *variables_and_units: dict | None = None*, *convert_calendar_kwargs: dict | None = None*, *missing_by_var: dict | None = None*, *maybe_unstack_dict: dict | None = None*, *round_var: dict | None = None*, *common_attrs_only: dict | list[Dataset | str | PathLike] | None = None*, *common_attrs_open_kwargs: dict | None = None*, *attrs_to_remove: dict | None = None*, *remove_all_attrs_except: dict | None = None*, *add_attrs: dict | None = None*, *change_attr_prefix: str | None = None*, *to_level: str | None = None*) → Dataset

Clean up of the dataset.

It can:

- convert to the right units using xscen.finalize.change_units
- convert the calendar and interpolate over missing dates
- call the xscen.common.maybe_unstack function
- remove a list of attributes
- remove everything but a list of attributes
- add attributes
- change the prefix of the catalog attrs

in that order.

Parameters

- **ds** (*xr.Dataset*) – Input dataset to clean up
- **variables_and_units** (*dict, optional*) – Dictionary of variable to convert. eg. {'tasmax': 'degC', 'pr': 'mm d-1'}
- **convert_calendar_kwargs** (*dict, optional*) – Dictionary of arguments to feed to xclim.core.calendar.convert_calendar. This will be the same for all variables. If missing_by_vars is given, it will override the 'missing' argument given here. Eg. {target': default, 'align_on': 'random'}
- **missing_by_var** (*dict, optional*) – Dictionary where the keys are the variables and the values are the argument to feed the *missing* parameters of the xclim.core.calendar.convert_calendar for the given variable with the *convert_calendar_kwargs*. When the value of an entry is 'interpolate', the missing values will be filled with NaNs, then linearly interpolated over time.
- **maybe_unstack_dict** (*dict, optional*) – Dictionary to pass to xscen.common.maybe_unstack function. The format should be: {'coords': path_to_coord_file, 'rechunk': {'time': -1 }, 'stack_drop_nans': True}.
- **round_var** (*dict, optional*) – Dictionary where the keys are the variables of the dataset and the values are the number of decimal places to round to
- **common_attrs_only** (*dict, list of datasets, or list of paths, optional*) – Dictionnary of datasets or list of datasets, or path to NetCDF or Zarr files. Keeps only the global attributes that are the same for all datasets and generates a new id.
- **common_attrs_open_kwargs** (*dict, optional*) – Dictionary of arguments for xarray.open_dataset(). Used with common_attrs_only if given paths.

- **attrs_to_remove** (*dict, optional*) – Dictionary where the keys are the variables and the values are a list of the attrs that should be removed. For global attrs, use the key 'global'. The element of the list can be exact matches for the attributes name or use the same substring matching rules as intake_esm: - ending with a '*' means checks if the substring is contained in the string - starting with a '^' means check if the string starts with the substring. eg. {'global': ['unnecessary note', 'cell*'], 'tasmax': 'old_name'}
- **remove_all_attrs_except** (*dict, optional*) – Dictionary where the keys are the variables and the values are a list of the attrs that should NOT be removed, all other attributes will be deleted. If None (default), nothing will be deleted. For global attrs, use the key 'global'. The element of the list can be exact matches for the attributes name or use the same substring matching rules as intake_esm: - ending with a '*' means checks if the substring is contained in the string - starting with a '^' means check if the string starts with the substring. eg. {'global': ['necessary note', '^cat:'], 'tasmax': 'new_name'}
- **add_attrs** (*dict, optional*) – Dictionary where the keys are the variables and the values are a another dictionary of attributes. For global attrs, use the key 'global'. eg. {'global': {'title': 'amazing new dataset'}, 'tasmax': {'note': 'important info about tasmax'}}
- **change_attr_prefix** (*str, optional*) – Replace "cat:" in the catalog global attrs by this new string
- **to_level** (*str, optional*) – The processing level to assign to the output.

Returns

xr.Dataset – Cleaned up dataset

See also:

`xclim.core.calendar.convert_calendar`

`xscen.utils.`**`date_parser`**`(`*date: str | datetime | Timestamp | datetime | Period*, ***, *end_of_period: bool | str = False*, *out_dtype: str = 'datetime'*, *strtime_format: str = '%Y-%m-%d'*, *freq: str = 'H'*`)` → str | Period | Timestamp

Return a datetime from a string.

Parameters

- **date** (*str, cftime.datetime, pd.Timestamp, datetime.datetime, pd.Period*) – Date to be converted
- **end_of_period** (*bool or str*) – If 'Y' or 'M', the returned date will be the end of the year or month that contains the received date. If True, the period is inferred from the date's precision, but *date* must be a string, otherwise nothing is done.
- **out_dtype** (*str*) – Choices are 'datetime', 'period' or 'str'
- **strtime_format** (*str*) – If out_dtype=='str', this sets the strftime format
- **freq** (*str*) – If out_dtype=='period', this sets the frequency of the period.

Returns

pd.Timestamp, pd.Period, str – Parsed date

`xscen.utils.`**`get_cat_attrs`**`(`*ds: Dataset | DataArray | dict*, *prefix: str = 'cat:'*, *var_as_str=False*`)` → dict

Return the catalog-specific attributes from a dataset or dictionary.

Parameters

- **ds** (*xr.Dataset, dict*) – Dataset to be parsed. If a dictionary, it is assumed to be the attributes of the dataset (ds.attrs).

- **prefix** (*str*) – Prefix automatically generated by intake-esm. With xscen, this should be 'cat:'
- **var_as_str** (*bool*) – If True, 'variable' will be returned as a string if there is only one.

Returns
dict – Compilation of all attributes in a dictionary.

xscen.utils.**maybe_unstack**(*ds: Dataset*, *coords: str | None = None*, *rechunk: dict | None = None*, *stack_drop_nans: bool = False*) → Dataset

If stack_drop_nans is True, unstack and rechunk.

Parameters

- **ds** (*xr.Dataset*) – Dataset to unstack.
- **coords** (*str, optional*) – Path to a dataset containing the coords to unstack (and only those).
- **rechunk** (*dict, optional*) – If not None, rechunk the dataset after unstacking.
- **stack_drop_nans** (*bool*) – If True, unstack the dataset and rechunk it. If False, do nothing.

Returns
xr.Dataset – Unstacked dataset.

xscen.utils.**minimum_calendar**(**calendars*) → str

Return the minimum calendar from a list.

Uses the hierarchy: 360_day < noleap < standard < all_leap, and returns one of those names.

xscen.utils.**natural_sort**(*_list: list[str]*)

For strings of numbers. alternative to sorted() that detects a more natural order.

e.g. [r3i1p1, r1i1p1, r10i1p1] is sorted as [r1i1p1, r3i1p1, r10i1p1] instead of [r10i1p1, r1i1p1, r3i1p1]

xscen.utils.**publish_release_notes**(*style: str = 'md'*, *file: PathLike | StringIO | TextIO | None = None*, *changes: str | PathLike = None*) → str | None

Format release history in Markdown or ReStructuredText.

Parameters

- **style** (*{"rst", "md"}*) – Use ReStructuredText (*rst*) or Markdown (*md*) formatting. Default: Markdown.
- **file** (*{os.PathLike, StringIO, TextIO, None}*) – If provided, prints to the given file-like object. Otherwise, returns a string.
- **changes** (*{str, os.PathLike}, optional*) – If provided, manually points to the file where the changelog can be found. Assumes a relative path otherwise.

Returns
str, optional

Notes

This function exists solely for development purposes. Adapted from xclim.testing.utils.publish_release_notes.

xscen.utils.**stack_drop_nans**(*ds: Dataset*, *mask: DataArray*, ***, *new_dim: str = 'loc'*, *to_file: str | None = None*) → Dataset

Stack dimensions into a single axis and drops indexes where the mask is false.

Parameters

- **ds** (*xr.Dataset*) – A dataset with the same coords as *mask*.
- **mask** (*xr.DataArray*) – A boolean DataArray with True on the points to keep. Mask will be loaded within this function.
- **new_dim** (*str*) – The name of the new stacked dim.
- **to_file** (*str, optional*) – A netCDF filename where to write the stacked coords for use in *unstack_fill_nan*. If given a string with {shape} and {domain}, the formatting will fill them with the original shape of the dataset and the global attributes 'cat:domain'. If None (default), nothing is written to disk. It is recommended to fill this argument in the config. It will be parsed automatically. E.g.:

  utils:

  stack_drop_nans:
  to_file: /some_path/coords/coords_{domain}_{shape}.nc

  unstack_fill_nan:
  coords: /some_path/coords/coords_{domain}_{shape}.nc

Returns

xr.Dataset – Same as *ds*, but all dimensions of mask have been stacked to a single *new_dim*. Indexes where mask is False have been dropped.

See also:

unstack_fill_nan
The inverse operation.

xscen.utils.**standardize_periods**(*periods: list[str] | list[list[str]] | None*, *multiple: bool = True*) → list[str] | list[list[str]] | None

Reformats the input to a list of strings, ['start', 'end'], or a list of such lists.

Parameters

- **periods** (*list of str or list of lists of str, optional*) – The period(s) to standardize. If None, return None.
- **multiple** (*bool*) – If True, return a list of periods, otherwise return a single period.

xscen.utils.**translate_time_chunk**(*chunks: dict*, *calendar: str*, *timesize*) → dict

Translate chunk specification for time into a number.

-1 translates to *timesize* 'Nyear' translates to N times the number of days in a year of calendar *calendar*.

xscen.utils.**unstack_dates**(*ds: Dataset*, *seasons: dict[int, str] | None = None*, *new_dim: str = 'season'*, *winter_starts_year: bool = False*)

Unstack a multi-season timeseries into a yearly axis and a season one.

Parameters

- **ds** (*xr.Dataset or DataArray*) – The xarray object with a "time" coordinate. Only supports monthly or coarser frequencies. The time axis must be complete and regular (*xr.infer_freq(ds.time)* doesn't fail).
- **seasons** (*dict, optional*) – A dictionary from month number (as int) to a season name. If not given, it is guessed from the time coord's frequency. See notes.
- **new_dim** (*str*) – The name of the new dimension.
- **winter_starts_year** (*bool*) – If True, the year of winter (DJF) is built from the year of January, not December. i.e. DJF made from [Dec 1980, Jan 1981, and Feb 1981] will be associated with the year 1981, not 1980.

Returns

xr.Dataset or DataArray – Same as ds but the time axis is now yearly (AS-JAN) and the seasons are along the new dimension.

Notes

When *season* is None, the inferred frequency determines the new coordinate:

- For MS, the coordinates are the month abbreviations in english (JAN, FEB, etc.)
- For ?QS-? and other ?MS frequencies, the coordinates are the initials of the months in each season. Ex: QS-DEC (with winter_starts_year=True) : DJF, MAM, JJA, SON.
- For YS or AS-JAN, the new coordinate has a single value of "annual".
- For ?AS-? frequencies, the new coordinate has a single value of "annual-{anchor}", were "anchor" is the abbreviation of the first month of the year. Ex: AS-JUL -> "annual-JUL".

xscen.utils.**unstack_fill_nan**(*ds: Dataset*, *, *dim: str = 'loc'*, *coords: str | PathLike | Sequence[str | PathLike] | dict | None = None*)

Unstack a Dataset that was stacked by `stack_drop_nans()`.

Parameters

- **ds** (*xr.Dataset*) – A dataset with some dims stacked by *stack_drop_nans*.
- **dim** (*str*) – The dimension to unstack, same as *new_dim* in *stack_drop_nans*.
- **coords** (*Sequence of strings, Mapping of str to array, str, optional*) – If a sequence : if the dataset has coords along *dim* that are not original dimensions, those original dimensions must be listed here. If a dict : a mapping from the name to the array of the coords to unstack If a str : a filename to a dataset containing only those coords (as coords). If given a string with {shape} and {domain}, the formatting will fill them with the original shape of the dataset (that should have been store in the attributes of the stacked dimensions) by *stack_drop_nans* and the global attributes 'cat:domain'. It is recommended to fill this argument in the config. It will be parsed automatically. E.g.:

  utils:

  stack_drop_nans:
  to_file: /some_path/coords/coords_{domain}_{shape}.nc

  unstack_fill_nan:
  coords: /some_path/coords/coords_{domain}_{shape}.nc

  If None (default), all coords that have *dim* a single dimension are used as the new dimensions/coords in the unstacked output. Coordinates will be loaded within this function.

Returns

xr.Dataset – Same as *ds*, but *dim* has been unstacked to coordinates in *coords*. Missing elements are filled according to the defaults of *fill_value* of `xarray.Dataset.unstack()`.

xscen.utils.**update_attr**(*ds: Dataset | DataArray*, *attr: str*, *new: str*, *others: Sequence[Dataset | DataArray] | None = None*, ***fmt*) → Dataset | DataArray

Format an attribute referencing itself in a translatable way.

Parameters

- **ds** (*Dataset or DataArray*) – The input object with the attribute to update.
- **attr** (*str*) – Attribute name.
- **new** (*str*) – New attribute as a template string. It may refer to the old version of the attribute with the "{attr}" field.
- **others** (*Sequence of Datasets or DataArrays*) – Other objects from which we can extract the attribute *attr*. These can be referenced as "{attrXX}" in *new*, where XX is the based-1 index of the other source in *others*. If they don't have the *attr* attribute, an empty string is sent to the string formatting. See notes.
- **fmt** – Other formatting data.

Returns

ds, but updated with the new version of *attr*, in each of the activated languages.

Notes

This is meant for constructing attributes by extending a previous version or combining it from different sources. For example, given a *ds* that has *long_name="Variability"*:

```
>>> update_attr(ds, "long_name", _("Mean of {attr}"))
```

Will update the "long_name" of *ds* with *long_name="Mean of Variability"*. The use of *_(...)* allows the detection of this string by the translation manager. The function will be able to add a translatable version of the string for each activated language, for example adding a *long_name_fr="Moyenne de Variabilité"* (assuming a *long_name_fr* was present on the initial *ds*).

If the new attribute is an aggregation from multiple sources, these can be passed in *others*.

```
>>> update_attr(
...     ds0,
...     "long_name",
...     _("Addition of {attr} and {attr1}, divided by {attr2}"),
...     others=[ds1, ds2],
... )
```

Here, *ds0* will have it's *long_name* updated with the passed string, where *attr1* is the *long_name* of *ds1* and *attr2* the *long_name* of *ds2*. The process will be repeated for each localized *long_name* available on *ds0*. For example, if *ds0* has a *long_name_fr*, the template string is translated and filled with the *long_name_fr* attributes of *ds0*, *ds1* and *ds2*. If the latter don't exist, the english version is used instead.

2.8 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.8.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/Ouranosinc/xscen/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

xscen could always use more documentation, whether as part of the official xscen docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/Ouranosinc/xscen/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.8.2 Get Started!

Note: If you are new to using GitHub and *git*, please read this guide first.

Warning: Anaconda Python users: Due to the complexity of some packages, the default dependency solver can take a long time to resolve the environment. Consider running the following commands in order to speed up the process:

```
$ conda install -n base conda-libmamba-solver
$ conda config --set solver libmamba
```

For more information, please see the following link: https://www.anaconda.com/blog/a-faster-conda-for-a-growing-community

Alternatively, you can use the mamba package manager, which is a drop-in replacement for `conda`. If you are already using *mamba*, replace the following commands with `mamba` instead of `conda`.

Ready to contribute? Here's how to set up `xscen` for local development.

1. Clone the repo locally:

   ```
   $ git clone git@github.com:Ouranosinc/xscen.git
   ```

2. Install your local copy into a development environment. You can create a new Anaconda development environment with:

   ```
   $ conda env create -f environment-dev.yml
   $ conda activate xscen-dev
   $ python -m pip install --editable ".[dev]"
   ```

   This installs `xscen` in an "editable" state, meaning that changes to the code are immediately seen by the environment.

3. As xscen was installed in editable mode, we also need to compile the translation catalogs manually:

   ```
   $ make translate
   ```

4. To ensure a consistent coding style, install the `pre-commit` hooks to your local clone:

   ```
   $ pre-commit install
   ```

   On commit, `pre-commit` will check that `black`, `blackdoc`, `isort`, `flake8`, and `ruff` checks are passing, perform automatic fixes if possible, and warn of violations that require intervention. If your commit fails the checks initially, simply fix the errors, re-add the files, and re-commit.

   You can also run the hooks manually with:

   ```
   $ pre-commit run -a
   ```

   If you want to skip the `pre-commit` hooks temporarily, you can pass the `--no-verify` flag to *$ git commit*.

5. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

Now you can make your changes locally.

6. When you're done making changes, we **strongly** suggest running the tests in your environment or with the help of `tox`:

```
$ python -m pytest
# Or, to run multiple build tests
$ tox
```

Alternatively, you can run the tests using *make*:

```
$ make lint
$ make test
```

Running *make lint* and *make test* demands that your runtime/dev environment have all necessary development dependencies installed.

> **Warning:** Due to some dependencies only being available via Anaconda/conda-forge or built from source, *tox*-based testing will only work if ESMF is available in your system path. This also requires that the *ESMF_VERSION* environment variable (matching the version of ESMF installed) be accessible within your shell as well (e.g.: *$ export ESMF_VERSION=8.5.0*).

7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

If `pre-commit` hooks fail, try re-committing your changes (or, if need be, you can skip them with *$ git commit –no-verify*).

8. Submit a Pull Request through the GitHub website.

9. When pushing your changes to your branch on GitHub, the documentation will automatically be tested to reflect the changes in your Pull Request. This build process can take several minutes at times. If you are actively making changes that affect the documentation and wish to save time, you can compile and test your changes beforehand locally with:

```
# To generate the html and open it in your browser
$ make docs
# To only generate the html
$ make autodoc
$ make -C docs html
# To simply test that the docs pass build checks
$ tox -e docs
```

Note: When building the documentation, the default behaviour is to evaluate notebooks ('nbsphinx_execute = "always"'), rather than simply parse the content ('nbsphinx_execute = "never"'). Due to their complexity, this can sometimes be a very computationally demanding task and should only be performed when necessary (i.e.: when the notebooks have been modified).

In order to speed up documentation builds, setting a value for the environment variable "SKIP_NOTEBOOKS" (e.g. "$ export SKIP_NOTEBOOKS=1") will prevent the notebooks from being evaluated on all subsequent "$ tox -e docs" or "$ make docs" invocations.

---

10. Once your Pull Request has been accepted and merged to the `main` branch, several automated workflows will be triggered:
    - The `bump-version.yml` workflow will automatically bump the patch version when pull requests are pushed to the `main` branch on GitHub. **It is not recommended to manually bump the version in your branch when merging (non-release) pull requests (this will cause the version to be bumped twice).**
    - *ReadTheDocs* will automatically build the documentation and publish it to the *latest* branch of *xscen* documentation website.
    - If your branch is not a fork (ie: you are a maintainer), your branch will be automatically deleted.

You will have contributed your first changes to `xscen`!

Translating xscen

If your additions to `xscen` play with plain text attributes like "long_name" or "description", you should also provide French translations for those fields. To manage translations, xscen uses python's `gettext` with the help of `babel`.

To update an attribute while enabling translation, use `utils.add_attr()` instead of a normal set-item. For example:

```
ds.attrs["description"] = "The English description"
```

becomes:

```
from xscen.utils import add_attr


def _(s):
    return s


add_attr(ds, "description", _("English description of {a}"), a="var")
```

See also `update_attr()` for the special case where an attribute is updated using its previous version.

Once the code is implemented and translatable strings are marked as such, we need to extract them and catalog them in the French translation map. From the root directory of xscen, run:

```
$ make findfrench
```

Then go edit `xscen/xscen/data/fr/LC_MESSAGES/xscen.po` with the correct French translations. Finally, running:

```
$ make translate
```

This will compile the edited catalogs, allowing python to detect and use them.

2.8.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests and should aim to provide code coverage for all new lines of code. You can use the `--cov-report html --cov xscen` flags during the call to `pytest` to generate an HTML report and analyse the current test coverage.
2. If the pull request adds functionality, the docs should also be updated. Put your new functionality into a function with a docstring, and add the feature to the list in `README.rst`.
3. The pull request should not break the templates.
4. The pull request should work for Python 3.9, 3.10, and 3.11. Check that the tests pass for all supported Python versions.

2.8.4 Tips

To run a subset of tests:

```
$ pytest tests.test_xscen
```

To run specific code style checks:

```
$ black --check xscen tests
$ isort --check xscen tests
$ blackdoc --check xscen docs
$ ruff xscen tests
$ flake8 xscen tests
```

To get `black`, `isort ``blackdoc`, `ruff`, and `flake8` (with plugins `flake8-alphabetize` and `flake8-rst-docstrings`) simply install them with *pip* (or *conda*) into your environment.

2.8.5 Versioning/Tagging

A reminder for the **maintainers** on how to deploy. This section is only relevant when producing a new point release for the package.

Warning: It is important to be aware that any changes to files found within the `xscen` folder (with the exception of `xscen/__init__.py`) will trigger the `bump-version.yml` workflow. Be careful not to commit changes to files in this folder when preparing a new release.

1. Create a new branch from *main* (e.g. *release-0.2.0*).
2. Update the *CHANGES.rst* file to change the *Unreleased* section to the current date.
3. Bump the version in your branch to the next version (e.g. *v0.1.0 -> v0.2.0*):

```
.. code-block:: shell

    $ bump-my-version bump minor # In most cases, we will be releasing a minor␣
↪version
    $ git push
```

4. Create a pull request from your branch to *main*.

5. Once the pull request is merged, create a new release on GitHub. On the main branch, run:

```
$ git tag v0.2.0
$ git push --tags
```

   This will trigger a GitHub workflow to build the package and upload it to TestPyPI. At the same time, the GitHub workflow will create a draft release on GitHub. Assuming that the workflow passes, the final release can then be published on GitHub by finalizing the draft release.

6. Once the release is published, the *publish-pypi.yml* workflow will go into an *awaiting approval* mode on Github Actions. Only authorized users may approve this workflow (notifications will be sent) to trigger the upload to PyPI.

Warning: Uploads to PyPI can **never** be overwritten. If you make a mistake, you will need to bump the version and re-release the package. If the package uploaded to PyPI is broken, you should modify the GitHub release to mark the package as broken, as well as yank the package (mark the version "broken") on PyPI.

2.8.6 Packaging

When a new version has been minted (features have been successfully integrated test coverage and stability is adequate), maintainers should update the `pip`-installable package (wheel and source release) on PyPI as well as the binary on conda-forge.

The simple approach

The simplest approach to packaging for general support (pip wheels) requires the following packages installed:

- build
- setuptools
- twine
- wheel

From the command line on your Linux distribution, simply run the following from the clone's main dev branch:

```
# To build the packages (sources and wheel)
$ python -m build --sdist --wheel

# To upload to PyPI
$ twine upload dist/*
```

2.9 Credits

2.9.1 Development Lead

- Gabriel Rondeau-Genesse <rondeau-genesse.gabriel@ouranos.ca> @RondeauG

2.9.2 Co-Developers

- Pascal Bourgault <bourgault.pascal@ouranos.ca> @aulemahal
- Juliette Lavoie <lavoie.juliette@ouranos.ca> @juliettelavoie
- Trevor James Smith <smith.trevorj@ouranos.ca> @Zeitsperre

2.9.3 Contributors

- Travis Logan <logan.travis@ouranos.ca> @tlogan2000
- Louis-Philippe Caron <caron.louis-philippe@ouranos.ca>
- Sarah Gammon <gammon.sarah@ouranos.ca> @SarahG-579462
- Yannick Rousseau
- Marco Braun <Braun.Marco@ouranos.ca> @vindelico
- Sarah-Claude Bourdeau-Goulet <bourdeau-goulet.sarah-claude@ouranos.ca> @sarahclaude

2.10 Changelog

2.10.1 v0.8.0 (unreleased)

Contributors to this version: Gabriel Rondeau-Genesse (@RondeauG), Pascal Bourgault (@aulemahal), Juliette Lavoie (@juliettelavoie), Sarah-Claude Bourdeau-Goulet (@sarahclaude), Trevor James Smith (@Zeitsperre), Marco Braun (@vindelico).

Announcements

- *xscen* now adheres to PEPs 517/518/621 using the *setuptools* and *setuptools-scm* backend for building and packaging. (PR/292).

New features and enhancements

- New function `xscen.indicators.select_inds_for_avail_vars` to filter the indicators that can be calculated with the variables available in a `xarray.Dataset`. (PR/291).
- Replaced aggregation function `climatological_mean()` with `climatological_op()` offering more types of operations to aggregate over climatological periods. (PR/290)
- Added the ability to search for simulations that reach a given warming level. (PR/251).
- `xs.spatial_mean` now accepts the `region="global"` keyword to perform a global average (GH/94, PR/260).
- `xs.spatial_mean` with `method='xESMF'` will also automatically segmentize polygons (down to a 1° resolution) to ensure a correct average (PR/260).
- Added documentation for *require_all_on* in *search_data_catalogs*. (PR/263).
- `xs.save_to_table` and `xs.io.to_table` to transform datasets and arrays to DataFrames, but with support for multi-columns, multi-sheets and localized table of content generation.
- Better `xs.extract.resample` : support for weighted resampling operations when starting with frequencies coarser than daily and missing timesteps/values handling. (GH/80, GH/93, PR/265).

- New argument `attribute_weights` to `generate_weights` to allow for custom weights. (PR/252).
- `xs.io.round_bits` to round floating point variable up to a number of bits, allowing for a better compression. This can be combined with the saving step through argument `"bitround"` of `save_to_netcdf` and `save_to_zarr`. (PR/266).
- Added annual global tas timeseries for CMIP6's models CMCC-ESM2 (ssp245, ssp370, ssp585), EC-Earth3-CC (ssp245, ssp585), KACE-1-0-G (ssp245, ssp370, ssp585) and TaiESM1 (ssp245, ssp370). Moved global tas database to a netCDF file. (GH/268, PR/270).
- Implemented support for multiple levels and models in `xs.subset_warming_level`. Better support for *DataArray* and *DataFrame* in `xs.get_warming_level`. (PR/270).
- Added the ability to directly provide an ensemble dataset to `xs.ensemble_stats`. (PR/299).
- Added support in `xs.ensemble_stats` for the new robustness-related functions available in *xclim*. (PR/299).
- New function `xs.ensembles.get_partition_input` (PR/289).

Breaking changes

- `climatological_mean()` has been replaced with `climatological_op()` and will be abandoned in a future version. (PR/290)
- `experiment_weights` argument in `generate_weights` was renamed to `balance_experiments`. (PR/252).
- New argument `attribute_weights` to `generate_weights` to allow for custom weights. (PR/252).
- For a sequence of models, the output of `xs.get_warming_level` is now a list. Revert to a dictionary with `output='selected'` (PR/270).
- The global average temperature database is now a netCDF, custom databases must follow the same format (PR/270).

Bug fixes

- Fixed a bug in `xs.search_data_catalogs` when searching for fixed fields and specific experiments/members. (PR/251).
- Fixed a bug in the documentation build configuration that prevented stable/latest and tagged documentation builds from resolving on ReadTheDocs. (PR/256).
- Fixed `get_warming_level` to avoid incomplete matches. (PR/269).
- *search_data_catalogs* now eliminates anything that matches any entry in *exclusions*. (GH/275, PR/280).
- Fixed a bug in `xs.scripting.save_and_update` where `build_path_kwargs` was ignored when trying to guess the file format. (PR/282).
- Add a warning to `xs.extract._dispatch_historical_to_future`. (GH/286, PR/287).
- Modify use_cftime for the calendar conversion in `to_dataset`. (GH/303, PR/289).

Internal changes

- Continued work on adding tests. (PR/251).
- Fixed *pre-commit*'s *pretty-format-json* hook so that it ignores notebooks. (PR/254).
- Fixed the labeler so docs/CI isn't automatically added for contributions by new collaborators. (PR/254).
- Made it so that *tests* are no longer treated as an installable package. (PR/248).
- Renamed the pytest marker from `requires_docs` to `requires_netcdf`. (PR/248).
- Included the documentation in the source distribution, while excluding the NetCDF files. (PR/248).
- Reduced the size of the files in `/docs/notebooks/samples` and changed the notebooks and tests accordingly. (GH/247, PR/248).
- Added a new *xscen.testing* module with the *datablock_3d* function previously located in `/tests/conftest.py`. (PR/248).
- New function *xscen.testing.fake_data* to generate fake data for testing. (PR/248).
- xESMF 0.8 Regridder and SpatialAverager argument `out_chunks` is now accepted by `xs.regrid_dataset` and `xs.spatial_mean`. (PR/260).
- **Testing, Packaging, and CI adjustments. (PR/274):**
  - *xscen* builds now install in a *tox* environment with *conda*-provided *ESMF* in GitHub Workflows.
  - *tox* now offers a method for installing esmpy from a tag/branch (via ESMF_VERSION environment variable).
  - *$ make translate* is now called on ReadTheDocs and within *tox*.
  - Linters are now called by order of most common failures first, to speed up the CI.
  - *Manifest.in* is much more specific about what is installed.
  - Re-adds a dev recipe to the *setup.py*.
- Multiple improvements to the docstrings and type annotations. (PR/282).
- *pip check* in conda builds in GitHub workflows have been temporarily set to always pass. (PR/288).
- **The *cookiecutter* template has been updated to the latest commit via *cruft*. (PR/292):**
  - *setup.py* has been mostly hollowed-out, save for the *babel*-related translation function.
  - *pyproject.toml* has been added, with most package configurations migrated into it.
  - *HISTORY.rst* has been renamed to *CHANGES.rst*.
  - *actions-version-updater.yml* has been added to automate the versioning of the package.
  - *pre-commit* hooks have been updated to the latest versions; *check-toml* and *toml-sort* have been added to cleanup the *pyproject.toml* file, and *check-json-schema* has been added to ensure GitHub and ReadTheDocs workflow files are valid.
  - *ruff* has been added to the linting tools to replace most *flake8* and *pydocstyle* verifications.
  - *tox* builds are more pure Python environment/PyPI-friendly.
  - *xscen* now uses *Trusted Publishing* for TestPyPI and PyPI uploads.
- Linting checks now examine the testing folder, function complexity, and alphabetical order of *__all__* lists. (PR/292).
- `publish_release_notes` now uses better logic for finding and reformatting the *CHANGES.rst* file. (PR/292).

- `bump2version` version-bumping utility was replaced by `bump-my-version`. (PR/292).
- Documentation build checks no longer fail due to broken external links; Notebooks are now nested and numbered. (PR/304).

2.10.2 v0.7.1 (2023-08-23)

- Update dependencies by removing `pygeos`, pinning `shapely>=2` and `intake-esm>=2023.07.07` as well as other small fixes to the environment files. (PR/243).
- Fix `xs.aggregate.spatial_mean` with method `cos-lat` when the data is on a rectilinear grid. (PR/243).

Internal changes

- Added a workflow that removes obsolete GitHub Workflow caches from merged pull requests. (PR/250).
- Added a workflow to perform automated labeling of pull requests, dependent on the files changed. (PR/250).

2.10.3 v0.7.0 (2023-08-22)

Contributors to this version: Gabriel Rondeau-Genesse (@RondeauG), Pascal Bourgault (@aulemahal), Trevor James Smith (@Zeitsperre), Juliette Lavoie (@juliettelavoie), Marco Braun (@vindelico).

Announcements

- Dropped support for Python 3.8, added support for 3.11. (PR/199, PR/222).
- *xscen* is now available on conda-forge, and can be installed with `conda install -c conda-forge xscen`. (PR/241)

New features and enhancements

- *xscen* now tracks code coverage using coveralls. (PR/187).
- New function *get_warming_level* to search within the IPCC CMIP global temperatures CSV without requiring data. (GH/208, PR/210).
- File re-structuration from catalogs with `xscen.catutils.build_path`. (PR/205, PR/237).
- New scripting functions *save_and_update* and *move_and_delete*. (PR/214).
- Spatial dimensions can be generalized as X/Y when rechunking and will be mapped to rlon/rlat or lon/lat accordingly. (PR/221).
- New argument *var_as_string* for *get_cat_attrs* to return variable names as strings. (PR/233).
- New argument *copy* for *move_and_delete*. (PR/233).
- New argument *restrict_year* for *compute_indicators*. (PR/233).
- Add more comments in the template. (PR/233, GH/232).
- `generate_weights` now allows to split weights between experiments, and make them vary along the time/horizon axis. (GH/108, PR/231).
- New independence_level, *institution*, added to `generate_weights`. (PR/231).
- Updated `produce_horizon` so it can accept multiple periods or warming levels. (PR/231, PR/240).

- Add more comments in the template. (PR/233, PR/235, GH/232).
- New function `diagnostics.health_checks` that can perform multiple checkups on a dataset. (PR/238).

Breaking changes

- Columns `date_start` and `date_end` now use a `datetime64[ms]` dtype. (PR/222).
- The default output of `date_parser` is now `pd.Timestamp` (`output_dtype='datetime'`). (PR/222).
- `date_parser(date, end_of_period=True)` has time "23:59:59", instead of "23:00". (PR/222, PR/237).
- `driving_institution` was removed from the "default" xscen columns. (PR/222).
- Folder parsing utilities (`parse_directory`) moved to `xscen.catutils`. Signature changed : `globpattern` removed, `dirglob` added, new `patterns` specifications. See doc for all changes. (PR/205).
- `compute_indicators` now returns all outputs produced by indicators with multiple outputs (such as *rain_season*). (PR/228).
- In `generate_weights`, independence_level *all* was renamed *model*. (PR/231).
- In response to a bugfix, results for `generate_weights(independence_level='GCM')` are significantly altered. (GH/230, PR/231).
- Legacy support for *stats_kwargs* in `ensemble_stats` was dropped. (PR/231).
- *period* in `produce_horizon` has been deprecated and replaced with *periods*. (PR/231).
- Some automated *to_level* were updated to reflect more recent changes. (PR/231).
- Removed `diagnostics.fix_unphysical_values`. (PR/238).

Bug fixes

- Fix bug in `unstack_dates` with seasonal climatological mean. (GH/202, PR/202).
- Added NotImplemented errors when trying to call *climatological_mean* and *compute_deltas* with daily data. (PR/187).
- Minor documentation fixes. (GH/223, PR/225).
- Fixed a bug in `unstack_dates` where it failed for anything other than seasons. (PR/228).
- `cleanup` with *common_attrs_only* now works even when no *cat* attribute is present in the datasets. (PR/231).

Internal changes

- Removed the pin on xarray's version. (GH/175, PR/199).
- Folder parsing utilities now in pure python, platform independent. New dependency `parse`. (PR/205).
- Updated ReadTheDocs configuration to prevent `--eager` installation of xscen (PR/209).
- Implemented a template to be used for unit tests. (PR/187).
- Updated GitHub Actions to remove deprecation warnings. (PR/187).
- Updated the cookiecutter used to generate boilerplate documentation and code via *cruft*. (PR/212).
- A few changes to *subset_warming_level* so it doesn't need *driving_institution*. (PR/215).
- Added more tests. (PR/228).

- In `compute_indicators`, the logic to manage indicators returning multiple outputs was simplified. (PR/228).

2.10.4 v0.6.0 (2023-05-04)

Contributors to this version: Trevor James Smith (@Zeitsperre), Juliette Lavoie (@juliettelavoie), Pascal Bourgault (@aulemahal), Gabriel Rondeau-Genesse (@RondeauG).

Announcements

- *xscen* is now offered as a conda package available through Anaconda.org. Refer to the installation documentation for more information. (GH/149, PR/171).
- Deprecation: Release 0.6.0 of *xscen* will be the last version to support `xscen.extract.clisops_subset`. Use `xscen.spatial.subset` instead. (PR/182, PR/184).
- Deprecation: The argument *region*, used in multiple functions, has been slightly reformatted. Release 0.6.0 of *xscen* will be the last version to support the old format. (GH/99, GH/101, PR/184).

New features and enhancements

- New 'cos-lat' averaging in *spatial_mean*. (GH/94, PR/125).
- Support for computing anomalies in *compute_deltas*. (PR/165).
- Add function *diagnostics.measures_improvement_2d*. (PR/167).
- Add function `regrid.create_bounds_rotated_pole` and automatic use in `regrid_dataset` and `spatial_mean`. This is temporary, while we wait for a functionning method in `cf_xarray`. (PR/174, GH/96).
- Add `spatial` submodule with functions `creep_weights` and `creep_fill` for filling NaNs using neighbours. (PR/174).
- Allow passing `GeoDataFrame` instances in `spatial_mean`'s `region` argument, not only geospatial file paths. (PR/174).
- Allow searching for periods in *catalog.search*. (GH/123, PR/170).
- Allow searching and extracting multiple frequencies for a given variable. (GH/168, PR/170).
- New masking feature in `extract_dataset`. (GH/180, PR/182).
- New function `xs.spatial.subset` to replace `xs.extract.clisops_subset` and add method "sel". (GH/180, PR/182).
- Add long_name attribute to diagnostics. (PR/189).
- Added a new YAML-centric notebook (GH/8, PR/191).
- New `utils.standardize_periods` to standardize that argument across multiple functions. (GH/87, PR/192).
- New *coverage_kwargs* argument added to `search_data_catalogs` to allow modifying the default values of `subset_file_coverage`. (GH/87, PR/192).

Breaking changes

- 'mean' averaging has been deprecated in *spatial_mean*. (PR/125).
- 'interp_coord' has been renamed to 'interp_centroid' in *spatial_mean*. (PR/125).
- The 'datasets' dimension of the output of `diagnostics.measures_heatmap` is renamed 'realization'. (PR/167).
- *_subset_file_coverage* was renamed *subset_file_coverage* and moved to `catalog.py` to prevent circular imports. (PR/170).
- *extract_dataset* doesn't fail when a variable is in the dataset, but not *variables_and_freqs*. (PR/185).
- The argument *period*, used in multiple function, is now always a single list, while *periods* is more flexible. (GH/87, PR/192).
- The parameters *reference_period* and *simulation_period* of `xscen.train` and `xscen.adjust` were renamed *period/periods* to respect the point above. (GH/87, PR/192).

Bug fixes

- Forbid pandas v1.5.3 in the environment files, as the linux conda build breaks the data catalog parser. (GH/161, PR/162).
- Only return requested variables when using `DataCatalog.to_dataset`. (PR/163).
- `compute_indicators` no longer crashes if less than 3 timesteps are produced. (PR/125).
- *xarray* is temporarily pinned below v2023.3.0 due to an API-breaking change. (GH/175, PR/173).
- *xscen.utils.unstack_fill_nan `* can now handle datasets that have non dimension coordinates. (GH/156, PR/175).
- *extract_dataset* now skips a simulation way earlier if the frequency doesn't match. (PR/170).
- *extract_dataset* now correctly tries to extract in reverse timedelta order. (PR/170).
- *compute_deltas* no longer creates all NaN values if the input dataset is in a non-standard calendar. (PR/188).

Internal changes

- *xscen* now manages packaging for PyPi and TestPyPI via GitHub workflows. (PR/159).
- Pre-load coordinates in `extract.clisops_subset` (PR/163).
- Minimal documentation for templates. (PR/163).
- *xscen* is now indexed in Zenodo, under the *ouranos* community of projects. (PR/164).
- Added a few relevant Shields to the README.rst. (PR/164).
- Better warning messages in `_subset_file_coverage` when coverage is insufficient. (PR/125).
- The top-level Makefile now includes a *linkcheck* recipe, and the ReadTheDocs configuration no longer reinstalls the *llvmlite* compiler library. (PR/173).
- The checkups on coverage and duplicates can now be skipped in *subset_file_coverage*. (PR/170).
- Changed the *ProjectCatalog* docstrings to make it more obvious that it needs to be created empty. (GH/99, PR/184).
- Added parse_config to *creep_fill*, *creep_weights*, and *reduce_ensemble* (PR/191).

2.10.5 v0.5.0 (2023-02-28)

Contributors to this version: Gabriel Rondeau-Genesse (@RondeauG), Juliette Lavoie (@juliettelavoie), Trevor James Smith (@Zeitsperre), Sarah Gammon (@SarahG-579462) and Pascal Bourgault (@aulemahal).

New features and enhancements

- Possibility of excluding variables read from file from the catalog produced by `parse_directory`. (PR/107).
- New functions `extract.subset_warming_level` and `aggregate.produce_horizon`. (PR/93).
- add *round_var* to *xs.clean_up*. (PR/93).
- New "timeout_cleanup" option for `save_to_zarr`, which removes variables that were in the process of being written when receiving a `TimeoutException`. (PR/106).
- New `scripting.skippable` context, allowing the use of CTRL-C to skip code sections. (PR/106).
- Possibility of fields with underscores in the patterns of `parse_directory`. (PR/111).
- New `utils.show_versions` function for printing or writing to file the dependency versions of *xscen*. (GH/109, PR/112).
- Added previously private notebooks to the documentation. (PR/108).
- Notebooks are now tested using *pytest* with *nbval*. (PR/108).
- New `restrict_warming_level` argument for `extract.search_data_catalogs` to filter dataset that are not in the warming level csv. (GH/105, PR/138).
- Set configuration value programmatically through `CONFIG.set`. (PR/144).
- New `to_dataset` method on `DataCatalog`. The same as `to_dask`, but exposing more aggregation options. (PR/147).
- New templates folder with one general template. (GH/151, PR/158).

Breaking changes

- Functions that are called internally can no longer parse the configuration. (PR/133).

Bug fixes

- **`clean_up` now converts the calendar of variables that use "interpolate" in "missing_by_var" at the same time.**
  - Hence, when it is a conversion from a 360_day calendar, the random dates are the same for all of the these variables. (GH/102, PR/104).
- `properties_and_measures` no longer casts month coordinates to string. (PR/106).
- *search_data_catalogs* no longer crashes if it finds nothing. (GH/42, PR/92).
- Prevented fixed fields from being duplicated during *_dispatch_historical_to_future* (GH/81, PR/92).
- Added missing *parse_config* to functions in *reduce.py* (PR/92).
- Added deepcopy before *skipna* is popped in *spatial_mean* (PR/92).
- *subset_warming_level* now validates that the data exists in the dataset provided (GH/117, PR/119).
- Adapt *stack_drop_nan* for the newest version of xarray (2022.12.0). (GH/122, PR/126).

- Fix *stack_drop_nan* not working if intermediate directories don't exist (GH/128).
- Fixed a crash when *compute_indicators* produced fixed fields (PR/139).

Internal changes

- `compute_deltas` skips the unstacking step if there is no time dimension and cast object dimensions to string. (PR/9)
- Added the "2sem" frequency to the translations CVs. (PR/111).
- Skip files we can't read in `parse_directory`. (PR/111).
- Fixed non-numpy-standard Docstrings. (PR/108).
- Added more metadata to package description on PyPI. (PR/108).
- Faster `search_data_catalogs` and `extract_dataset` through a faster `DataCatalog.unique`, date parsing and a rewrite of the `ensure_correct_time` logic. (PR/127).
- The `search_data_catalogs` function now accepts *str* or *pathlib.Path* variables (in addition to lists of either data type) for performing catalog lookups. (PR/121).
- *produce_horizons* now supports fixed fields (PR/139).
- Rewrite of `unstack_dates` for better performance with dask arrays. (PR/144).

2.10.6 v0.4.0 (2022-09-28)

Contributors to this version: Gabriel Rondeau-Genesse (@RondeauG), Juliette Lavoie (@juliettelavoie), Trevor James Smith (@Zeitsperre) and Pascal Bourgault (@aulemahal).

New features and enhancements

- New functions `diagnostics.properties_and_measures`, `diagnostics.measures_heatmap` and `diagnostics.measures_improvement`. (GH/5, PR/54).
- Add argument *resample_methods* to *xs.extract.resample*. (GH/57, PR/57)
- Added a ReadTheDocs configuration to expose public documentation. (GH/65, PR/66).
- `xs.utils.stack_drop_nans/ xs.utils.unstack_fill_nan` will now format the *to_file/coords* string to add the domain and the shape. (GH/59, PR/67).
- New unstack_dates function to "extract" seasons or months from a timeseries. (PR/68).
- Better spatial_mean for cases using xESMF and a shapefile with multiple polygons. (PR/68).
- **Yet more changes to parse_directory: (PR/68).**
  - Better parallelization by merging the finding and name-parsing step in the same dask tree.
  - Allow cvs for the variable columns.
  - Fix parsing the variable names from datasets.
  - Sort the variables in the tuples (for a more consistent output)
- In extract_dataset, add option `ensure_correct_time` to ensure the time coordinate matches the expected freq. Ex: monthly values given on the 15th day are moved to the 1st, as expected when asking for "MS". (:issue: *53*).
- **In regrid_dataset: (PR/68).**

  - Allow passing skipna to the regridder kwargs.
  - Do not fail for any grid mapping problem, includin if a grid_mapping attribute mentions a variable that doesn't exist.

- Default email sent to the local user. (PR/68).
- Special accelerated pathway for parsing catalogs with all dates within the datetime64[ns] range. (PR/75).
- New functions `reduce_ensemble` and `build_reduction_data` to support kkz and kmeans clustering. (GH/4, PR/63).
- *ensemble_stats* can now loop through multiple statistics, support functions located in *xclim.ensembles._robustness*, and supports weighted realizations. (PR/63).
- New function *ensemble_stats.generate_weights* that estimates weights based on simulation metadata. (PR/63).
- New function *catalog.unstack_id* to reverse-engineer IDs. (PR/63).
- *generate_id* now accepts Datasets. (PR/63).
- Add *rechunk* option to *properties_and_measures* (PR/76).
- Add *create* argument to *ProjectCatalog* (GH/11, PR/77).
- Add percentage deltas to *compute_deltas* (GH/82, PR/90).

Breaking changes

- *statistics / stats_kwargs* have been changed/eliminated in *ensemble_stats*, respectively. (PR/63).

Bug fixes

- Add a missing dependencies to the env (*pyarrow*, for faster string handling in catalogs). (PR/68).
- Allow passing `compute=False` to *save_to_zarr*. (PR/68).

Internal changes

- Small bugfixes in *aggregate.py*. (PR/55, PR/56).
- Default method of *xs.extract.resample* now depends on frequency. (GH/57, PR/58).
- Bugfix for *_restrict_by_resolution* with CMIP6 datasets (PR/71).
- More complete check of coverage in `_subset_file_coverage`. (GH/70, PR/72)
- The code that performs `common_attrs_only` in *ensemble_stats* has been moved to *clean_up*. (PR/63).
- Removed the default `to_level` in *clean_up*. (PR/63).
- *xscen* now has an official logo. (PR/69).
- Use numpy max and min in *properties_and_measures* (PR/76).
- Cast catalog date_start and date_end to "%4Y-%m-%d %H:00" when writing to disk. (GH/83, PR/79)
- Skip test of coverage on the sum if the list of select files is empty. (PR/79)
- Added missing CMIP variable names in conversions.yml and added the ability to provide a custom file instead (GH/86, PR/88)
- Changed 'allow_conversion' and 'allow_resample' default to False in search_data_catalogs (GH/86, PR/88)

2.10.7 v0.3.0 (2022-08-23)

Contributors to this version: Gabriel Rondeau-Genesse (@RondeauG), Juliette Lavoie (@juliettelavoie), Trevor James Smith (@Zeitsperre) and Pascal Bourgault (@aulemahal).

New features and enhancements

- New function `clean_up` added. (GH/22, PR/25).
- *parse_directory*: Fixes to *xr_open_kwargs* and support for wildcards (*) in the directories. (PR/19).
- New function `xscen.ensemble.ensemble_stats` added. (GH/3, PR/28).
- New functions `spatial_mean`, `climatological_mean` and `deltas` added. (GH/4, PR/35).
- Add argument `intermediate_reg_grids` to `xscen.regridding.regrid`. (GH/34, PR/39).
- Add argument `moving_yearly_window` to `xscen.biasadjust.adjust`. (PR/39).
- Many adjustments to `parse_directory`: better wildcards (GH/24), allow custom columns, fastpaths for `parse_from_ds`, and more (PR/30).
- Documentation now makes better use of autodoc to generate package index. (PR/41).
- *periods* argument added to *compute_indicators* to support datasets with jumps in time (PR/35).

Breaking changes

- Patterns in `parse_directory` start at the end of the paths in `directories`. (PR/30).
- Argument `extension` of `parse_directory` has been renamed `globpattern`. (PR/30).
- **The `xscen` API and filestructure have been significantly refactored. (GH/40, PR/41). The following functions are available from the top-level:**
  - `adjust`, `train`, `ensemble_stats`, `clisops_subset`, `dispatch_historical_to_future`, `extract_dataset`, `resample`, `restrict_by_resolution`, `restrict_multimembers`, `search_data_catalogs`, `save_to_netcdf`, `save_to_zarr`, `rechunk`, `compute_indicators`, `regrid_dataset`, and `create_mask`.
- xscen now requires geopandas and shapely (PR/35).
- Following a change in intake-esm xscen now uses "cat:" to prefix the dataset attributes extracted from the catalog. All catalog-generated attributes should now be valid when saving to netCDF. (GH/13, PR/51).

Internal changes

- *parse_directory*: Fixes to *xr_open_kwargs*. (PR/19).
- Fix for indicators removing the 'time' dimension. (PR/23).
- Security scanning using CodeQL and GitHub Actions is now configured for the repository. (PR/21).
- Bumpversion action now configured to automatically augment the version number on each merged pull request. (PR/21).
- Add `align_on = 'year'` argument in bias adjustment converting of calendars. (PR/39).
- GitHub Actions using Ubuntu-22.04 images are now configured for running testing ensemble using *tox-conda*. (PR/44).

- *import xscen* smoke test is now run on all pull requests. (PR/44).
- Fix for *create_mask* removing attributes (PR/35).

2.10.8 v0.2.0 (first official release)

Contributors to this version: Gabriel Rondeau-Genesse (@RondeauG), Pascal Bourgault (@aulemahal), Trevor James Smith (@Zeitsperre), Juliette Lavoie (@juliettelavoie).

Announcements

- This is the first official release for xscen!

New features and enhancements

- Supports workflows with YAML configuration files for better transparency, reproducibility, and long-term backups.
- Intake_esm-based catalog to find and manage climate data.
- Climate dataset extraction, subsetting, and temporal aggregation.
- Calculate missing variables through Intake-esm's DerivedVariableRegistry.
- Regridding with xESMF.
- Bias adjustment with xclim.

Breaking changes

- N/A

Internal changes

- N/A

2.11 xscen

2.11.1 xscen package

A climate change scenario-building analysis framework, built with xclim/xarray.

`xscen.warning_on_one_line`(*message: str*, *category: Warning*, *filename: str*, *lineno: int*, *file=None*, *line=None*)

Monkeypatch Reformat warning so that *warnings.warn* doesn't mention itself.

Subpackages

xscen.xclim_modules package

xclim extension module.

Submodules

xscen.xclim_modules.conversions module

Conversion functions for when datasets are missing particular variables and that xclim doesn't already implement.

xscen.xclim_modules.conversions.**dtr**(*tasmin: DataArray*, *tasmax: DataArray*) → DataArray

DTR computed from tasmin and tasmax.

Dtr as tasmin subtracted from tasmax.

Parameters

- **tasmin** (*xr.DataArray*) – Daily minimal temperature.
- **tasmax** (*xr.DataArray*) – Daily maximal temperature.

Returns

xr.DataArray, K – Daily temperature range

xscen.xclim_modules.conversions.**precipitation**(*prsn: DataArray*, *prlp: DataArray*) → DataArray

Precipitation of all phases.

Compute the precipitation flux from all phases by adding solid and liquid precipitation.

Parameters

- **prsn** (*xr.DataArray*) – Solid precipitation flux.
- **prlp** (*xr.DataArray*) – Liquid precipitation flux.

Returns

xr.DataArray, [same as prsn] – Surface precipitation flux (all phases)

xscen.xclim_modules.conversions.**tasmax_from_dtr**(*dtr: DataArray*, *tasmin: DataArray*) → DataArray

Tasmax computed from DTR and tasmin.

Tasmax as dtr added to tasmin.

Parameters

- **dtr** (*xr.DataArray*) – Daily temperature range
- **tasmin** (*xr.DataArray*) – Daily minimal temperature.

Returns

xr.DataArray, [same as tasmin] – Daily maximum temperature

xscen.xclim_modules.conversions.**tasmin_from_dtr**(*dtr: DataArray*, *tasmax: DataArray*) → DataArray

Tasmin computed from DTR and tasmax.

Tasmin as dtr subtracted from tasmax.

Parameters

- **dtr** (*xr.DataArray*) – Daily temperature range

- **tasmax** (*xr.DataArray*) – Daily maximal temperature.

Returns

xr.DataArray, [same as tasmax] – Daily minimum temperature

Submodules

xscen.aggregate module

Functions to aggregate data over time and space.

xscen.aggregate.**climatological_mean**(*ds: Dataset*, ***, *window: int | None = None*, *min_periods: int | None = None*, *interval: int = 1*, *periods: list[str] | list[list[str]] | None = None*, *to_level: str | None = 'climatology'*) → Dataset

Compute the mean over 'year' for given time periods, respecting the temporal resolution of ds.

Parameters

- **ds** (*xr.Dataset*) – Dataset to use for the computation.
- **window** (*int, optional*) – Number of years to use for the time periods. If left at None and periods is given, window will be the size of the first period. If left at None and periods is not given, the window will be the size of the input dataset.
- **min_periods** (*int, optional*) – For the rolling operation, minimum number of years required for a value to be computed. If left at None and the xrfreq is either QS or AS and doesn't start in January, min_periods will be one less than window. If left at None, it will be deemed the same as 'window'.
- **interval** (*int*) – Interval (in years) at which to provide an output.
- **periods** (*list of str or list of lists of str, optional*) – Either [start, end] or list of [start, end] of continuous periods to be considered. This is needed when the time axis of ds contains some jumps in time. If None, the dataset will be considered continuous.
- **to_level** (*str, optional*) – The processing level to assign to the output. If None, the processing level of the inputs is preserved.

Returns

xr.Dataset – Returns a Dataset of the climatological mean, by calling climatological_op with option op=='mean'.

xscen.aggregate.**climatological_op**(*ds: Dataset*, ***, *op: str | dict = 'mean'*, *window: int | None = None*, *min_periods: int | float | None = None*, *stride: int = 1*, *periods: list[str] | list[list[str]] | None = None*, *rename_variables: bool = True*, *to_level: str = 'climatology'*, *horizons_as_dim: bool = False*) → Dataset

Perform an operation 'op' over time, for given time periods, respecting the temporal resolution of ds.

Parameters

- **ds** (*xr.Dataset*) – Dataset to use for the computation.
- **op** (*str or dict*) – Operation to perform over time. The operation can be any method name of xarray.core.rolling.DatasetRolling, 'linregress', or a dictionary. If 'op' is a dictionary, the key is the operation name and the value is a dict of kwargs accepted by the operation. While other operations are technically possible, the following are recommended and tested: ['max', 'mean', 'median', 'min', 'std', 'sum', 'var', 'linregress']. Operations beyond methods of xarray.core.rolling.DatasetRolling include:

- 'linregress' : Computes the linear regression over time, using scipy.stats.linregress and employing years as regressors. The output will have a new dimension 'linreg_param' with coordinates: ['slope', 'intercept', 'rvalue', 'pvalue', 'stderr', 'intercept_stderr'].

Only one operation per call is supported, so len(op)==1 if a dict.

- **window** (*int, optional*) – Number of years to use for the rolling operation. If left at None and periods is given, window will be the size of the first period. Hence, if periods are of different lengths, the shortest period should be passed first. If left at None and periods is not given, the window will be the size of the input dataset.
- **min_periods** (*int or float, optional*) – For the rolling operation, minimum number of years required for a value to be computed. If left at None and the xrfreq is either QS or AS and doesn't start in January, min_periods will be one less than window. Otherwise, if left at None, it will be deemed the same as 'window'. If passed as a float value between 0 and 1, this will be interpreted as the floor of the fraction of the window size.
- **stride** (*int*) – Stride (in years) at which to provide an output from the rolling window operation.
- **periods** (*list of str or list of lists of str, optional*) – Either [start, end] or list of [start, end] of continuous periods to be considered. This is needed when the time axis of ds contains some jumps in time. If None, the dataset will be considered continuous.
- **rename_variables** (*bool*) – If True, '_clim_{op}' will be added to variable names.
- **to_level** (*str, optional*) – The processing level to assign to the output. If None, the processing level of the inputs is preserved.
- **horizons_as_dim** (*bool*) – If True, the output will have 'horizon' and the frequency as 'month', 'season' or 'year' as dimensions and coordinates. The 'time' coordinate will be unstacked to horizon and frequency dimensions. Horizons originate from periods and/or windows and their stride in the rolling operation.

Returns

xr.Dataset – Dataset with the results from the climatological operation.

xscen.aggregate.**compute_deltas**(*ds: Dataset*, *reference_horizon: str | Dataset*, ***, *kind: str | dict = '+'*, *rename_variables: bool = True*, *to_level: str | None = 'deltas'*) → Dataset

Compute deltas in comparison to a reference time period, respecting the temporal resolution of ds.

Parameters

- **ds** (*xr.Dataset*) – Dataset to use for the computation.
- **reference_horizon** (*str or xr.Dataset*) – Either a YYYY-YYYY string corresponding to the 'horizon' coordinate of the reference period, or a xr.Dataset containing the climatological mean.
- **kind** (*str or dict*) – ['+', '/', '%'] Whether to provide absolute, relative, or percentage deltas. Can also be a dictionary separated per variable name.
- **rename_variables** (*bool*) – If True, '_delta_YYYY-YYYY' will be added to variable names.
- **to_level** (*str, optional*) – The processing level to assign to the output. If None, the processing level of the inputs is preserved.

Returns

xr.Dataset – Returns a Dataset with the requested deltas.

xscen.aggregate.**produce_horizon**(*ds: Dataset*, *indicators: str | PathLike | Sequence[Indicator] | Sequence[tuple[str, Indicator]] | module*, ***, *periods: list[str] | list[list[str]] | None = None*, *warminglevels: dict | None = None*, *to_level: str | None = 'horizons'*, *period: list | None = None*) → Dataset

Compute indicators, then the climatological mean, and finally unstack dates in order to have a single dataset with all indicators of different frequencies.

Once this is done, the function drops 'time' in favor of 'horizon'. This function computes the indicators and does an interannual mean. It stacks the season and month in different dimensions and adds a dimension *horizon* for the period or the warming level, if given.

Parameters

- **ds** (*xr.Dataset*) – Input dataset with a time dimension.
- **indicators** (*Union[str, os.PathLike, Sequence[Indicator], Sequence[Tuple[str, Indicator]], ModuleType]*) – Indicators to compute. It will be passed to the *indicators* argument of *xs.compute_indicators*.
- **periods** (*list of str or list of lists of str, optional*) – Either [start, end] or list of [start_year, end_year] for the period(s) to be evaluated. If both periods and warminglevels are None, the full time series will be used.
- **warminglevels** (*dict, optional*) – Dictionary of arguments to pass to *py:func:xscen.subset_warming_level*. If 'wl' is a list, the function will be called for each value and produce multiple horizons. If both periods and warminglevels are None, the full time series will be used.
- **to_level** (*str, optional*) – The processing level to assign to the output. If there is only one horizon, you can use "{wl}", "{period0}" and "{period1}" in the string to dynamically include that information in the processing level.

Returns

xr.Dataset – Horizon dataset.

xscen.aggregate.**spatial_mean**(*ds: Dataset*, *method: str*, ***, *spatial_subset: bool | None = None*, *call_clisops: bool | None = False*, *region: str | dict | None = None*, *kwargs: dict | None = None*, *simplify_tolerance: float | None = None*, *to_domain: str | None = None*, *to_level: str | None = None*) → Dataset

Compute the spatial mean using a variety of available methods.

Parameters

- **ds** (*xr.Dataset*) – Dataset to use for the computation.
- **method** (*str*) – 'cos-lat' will weight the area covered by each pixel using an approximation based on latitude. 'interp_centroid' will find the region's centroid (if coordinates are not fed through kwargs), then perform a .interp() over the spatial dimensions of the Dataset. The coordinate can also be directly fed to .interp() through the 'kwargs' argument below. 'xesmf' will make use of xESMF's SpatialAverager. This will typically be more precise, especially for irregular regions, but can be much slower than other methods.
- **spatial_subset** (*bool, optional*) – If True, xscen.spatial.subset will be called prior to the other operations. This requires the 'region' argument. If None, this will automatically become True if 'region' is provided and the subsetting method is either 'cos-lat' or 'mean'.
- **region** (*dict or str, optional*) – Description of the region and the subsetting method (required fields listed in the Notes). If method=='interp_centroid', this is used to find the

region's centroid. If method=='xesmf', the bounding box or shapefile is given to SpatialAverager. Can also be "global", for global averages. This is simply a shortcut for *{'name': 'global', 'method': 'bbox', 'lon_bnds' [-180, 180], 'lat_bnds': [-90, 90]}*.

- **kwargs** (*dict, optional*) – Arguments to send to either mean(), interp() or SpatialAverager(). For SpatialAverager, one can give *skipna* or *out_chunks* here, to be passed to the averager call itself.
- **simplify_tolerance** (*float, optional*) – Precision (in degree) used to simplify a shapefile before sending it to SpatialAverager(). The simpler the polygons, the faster the averaging, but it will lose some precision.
- **to_domain** (*str, optional*) – The domain to assign to the output. If None, the domain of the inputs is preserved.
- **to_level** (*str, optional*) – The processing level to assign to the output. If None, the processing level of the inputs is preserved.

Returns

xr.Dataset – Returns a Dataset with the spatial dimensions averaged.

Notes

'region' required fields:

name: str
Region name used to overwrite domain in the catalog.

method: str
['gridpoint', 'bbox', shape', 'sel']

tile_buffer: float, optional
Multiplier to apply to the model resolution. Only used if spatial_subset==True.

kwargs
Arguments specific to the method used.

See also:

`xarray.Dataset.mean`, `xarray.Dataset.interp`, `xesmf.SpatialAverager`

xscen.biasadjust module

Functions to train and adjust a dataset using a bias-adjustment algorithm.

`xscen.biasadjust.adjust`(*dtrain: Dataset*, *dsim: Dataset*, *periods: list[str] | list[list[str]]*, ***, *xclim_adjust_args: dict | None = None*, *to_level: str = 'biasadjusted'*, *bias_adjust_institution: str | None = None*, *bias_adjust_project: str | None = None*, *moving_yearly_window: dict | None = None*, *align_on: str | None = 'year'*) → Dataset

Adjust a simulation.

Parameters

- **dtrain** (*xr.Dataset*) – A trained algorithm's dataset, as returned by *train*.
- **dsim** (*xr.Dataset*) – Simulated timeseries, projected period.
- **periods** (*list of str or list of lists of str*) – Either [start, end] or list of [start, end] of the simulation periods to be adjusted (one at a time).

- **xclim_adjust_args** (*dict, optional*) – Dict of arguments to pass to the *.adjust* of the adjustment object.
- **to_level** (*str*) – The processing level to assign to the output. Defaults to 'biasadjusted'
- **bias_adjust_institution** (*str, optional*) – The institution to assign to the output.
- **bias_adjust_project** (*str, optional*) – The project to assign to the output.
- **moving_yearly_window** (*dict, optional*) – Arguments to pass to *xclim.sdba.construct_moving_yearly_window*. If not None, *construct_moving_yearly_window* will be called on dsim (and scen in xclim_adjust_args if it exists) before adjusting and *unpack_moving_yearly_window* will be called on the output after the adjustment. *construct_moving_yearly_window* stacks windows of the dataArray in a new 'movingwin' dimension. *unpack_moving_yearly_window* unpacks it to a normal time series.
- **align_on** (*str, optional*) – *align_on* argument for the fonction *xclim.core.calendar.convert_calendar*.

Returns

xr.Dataset – dscen, the bias-adjusted timeseries.

See also:

`xclim.sdba.adjustment.DetrendedQuantileMapping`, `xclim.sdba.adjustment.ExtremeValues`

xscen.biasadjust.**train**(*dref: Dataset*, *dhist: Dataset*, *var: str | list[str]*, *period: list[str]*, ***, *method: str = 'DetrendedQuantileMapping'*, *group: Grouper | str | dict | None = None*, *xclim_train_args: dict | None = None*, *maximal_calendar: str = 'noleap'*, *adapt_freq: dict | None = None*, *jitter_under: dict | None = None*, *jitter_over: dict | None = None*, *align_on: str | None = 'year'*) → Dataset

Train a bias-adjustment.

Parameters

- **dref** (*xr.Dataset*) – The target timeseries, on the reference period.
- **dhist** (*xr.Dataset*) – The timeseries to adjust, on the reference period.
- **var** (*str or list of str*) – Variable on which to do the adjustment. Currently only supports one variable.
- **period** (*list of str*) – [start, end] of the reference period
- **method** (*str*) – Name of the *sdba.TrainAdjust* method of xclim.
- **group** (*str or sdba.Grouper or dict, optional*) – Grouping information. If a string, it is interpreted as a grouper on the time dimension. If a dict, it is passed to *sdba.Grouper.from_kwargs*. Defaults to {"group": "time.dayofyear", "window": 31}.
- **xclim_train_args** (*dict*) – Dict of arguments to pass to the *.train* of the adjustment object.
- **maximal_calendar** (*str*) – Maximal calendar dhist can be. The hierarchy: 360_day < noleap < standard < all_leap. If dhist's calendar is higher than maximal calendar, it will be converted to the maximal calendar.
- **adapt_freq** (*dict, optional*) – If given, a dictionary of args to pass to the frequency adaptation function.
- **jitter_under** (*dict, optional*) – If given, a dictionary of args to pass to *jitter_under_thresh*.
- **jitter_over** (*dict, optional*) – If given, a dictionary of args to pass to *jitter_over_thresh*.

- **align_on** (*str, optional*) – *align_on* argument for the function *xclim.core.calendar.convert_calendar*.

Returns

xr.Dataset – Trained algorithm's data.

See also:

`xclim.sdba.adjustment.DetrendedQuantileMapping`, `xclim.sdba.adjustment.ExtremeValues`

xscen.catalog module

Catalog objects and related tools.

`xscen.catalog.COLUMNS = ['id', 'type', 'processing_level', 'bias_adjust_institution', 'bias_adjust_project', 'mip_era', 'activity', 'driving_model', 'institution', 'source', 'experiment', 'member', 'xrfreq', 'frequency', 'variable', 'domain', 'date_start', 'date_end', 'version', 'format', 'path']`

Official column names.

`class xscen.catalog.DataCatalog(*args, **kwargs)`

Bases: `esm_datastore`

A read-only intake_esm catalog adapted to xscen's syntax.

This class expects the catalog to have the columns listed in *xscen.catalog.COLUMNS* and it comes with default arguments for reading the CSV files (`xscen.catalog.csv_kwargs`). For example, all string columns (except *path*) are cast to a categorical dtype and the datetime columns are parsed with a special function that allows dates outside the conventional *datetime64[ns]* bounds by storing the data using `pandas.Period` objects.

Parameters

- ***args** (*str or os.PathLike or dict*) – Path to a catalog JSON file. If a dict, it must have two keys: 'esmcat' and 'df'. 'esmcat' must be a dict representation of the ESM catalog. 'df' must be a Pandas DataFrame containing content that would otherwise be in the CSV file.
- **check_valid** (*bool*) – If True, will check that all files in the catalog exist on disk and remove those that don't.
- **drop_duplicates** (*bool*) – If True, will drop duplicates in the catalog based on the 'id' and 'path' columns.
- ****kwargs** (*dict*) – Any other arguments are passed to intake_esm.esm_datastore.

See also:

`intake_esm.core.esm_datastore`

`check_valid()`

Verify that all files in the catalog exist on disk and remove those that don't.

If a file is a Zarr, it will also check that all variables are present and remove those that aren't.

`drop_duplicates(columns: list[str] | None = None)`

Drop duplicates in the catalog based on a subset of columns.

Parameters

columns (*list of str, optional*) – The columns used to identify duplicates. If None, 'id' and 'path' are used.

`exists_in_cat`(***columns*) → bool

Check if there is an entry in the catalogue corresponding to the arguments given.

Parameters
columns (Arguments that will be given to *catalog.search*)

Returns
bool – True if there is an entry in the catalogue corresponding to the arguments given.

`classmethod from_df`(*data: DataFrame | PathLike | Sequence[PathLike]*, *esmdata: PathLike | dict | None = None*, ***, *read_csv_kwargs: Mapping[str, Any] | None = None*, *name: str = 'virtual'*, ***intake_kwargs*)

Create a DataCatalog from one or more csv files.

Parameters

- **data** (*DataFrame or path or sequence of paths*) – A DataFrame or one or more paths to csv files.
- **esmdata** (*path or dict, optional*) – The "ESM collection data" as a path to a json file or a dict. If None (default), xscen's default `esm_col_data` is used.
- **read_csv_kwargs** (*dict, optional*) – Extra kwargs to pass to *pd.read_csv*, in addition to the ones in `csv_kwargs`.
- **name** (*str*) – If *metadata* doesn't contain it, a name to give to the catalog.

See also:

`pandas.read_csv`

`iter_unique`(**columns*)

Iterate over sub-catalogs for each group of unique values for all specified columns.

This is a generator that yields a tuple of the unique values of the current group, in the same order as the arguments, and the sub-catalog.

`search`(***columns*)

Modification of .search() to add the 'periods' keyword.

`to_dataset`(*concat_on: str | list[str] | None = None*, *create_ensemble_on: str | list[str] | None = None*, *ensemble_name: list[str] | None = None*, *calendar: str | None = 'standard'*, ***kwargs*) → Dataset

Open the catalog's entries into a single dataset.

Same as `to_dask()`, but with additional control over the aggregations. The dataset definition logic is left untouched by this method (by default: ["id", "domain", "processing_level", "xrfreq"]), except that newly aggregated columns are removed from the "id". This will override any "custom" id, ones not unstackable with *unstack_id()*.

Ensemble preprocessing logic is taken from `xclim.ensembles.create_ensemble()`. When *create_ensemble_on* is given, the function ensures all entries have the correct time coordinate according to *xrfreq*.

Parameters

- **concat_on** (*list of str or str, optional*) – A list of catalog columns over which to concat the datasets (in addition to 'time'). Each will become a new dimension with the column values as coordinates. Xarray concatenation rules apply and can be acted upon through *xarray_combine_by_coords_kwargs*.

- **create_ensemble_on** (*list of str or str, optional*) – The given column values will be merged into a new id-like "realization" column, which will be concatenated over. The given columns are removed from the dataset id, to remove them from the groupby_attrs logic. Xarray concatenation rules apply and can be acted upon through *xarray_combine_by_coords_kwargs*.
- **ensemble_name** (*list of strings, optional*) – If *create_ensemble_on* is given, this can be a subset of those column names to use when constructing the realization coordinate. If None, this will be the same as *create_ensemble_on*. The resulting coordinate must be unique.
- **calendar** (*str, optional*) – If *create_ensemble_on* is given, all datasets are converted to this calendar before concatenation. Ignored otherwise (default). If None, no conversion is done. *align_on* is always "date".
- **kwargs** – Any other arguments are passed to `to_dataset_dict()`. The *preprocess* argument cannot be used if *create_ensemble_on* is given.

Returns
`Dataset`

See also:

`intake_esm.core.esm_datastore.to_dataset_dict`, `intake_esm.core.esm_datastore.to_dask`, `xclim.ensembles.create_ensemble`

unique(*columns: str | Sequence[str] | None = None*)

Return a series of unique values in the catalog.

Parameters
columns (*str or sequence of str, optional*) – The columns to get unique values from. If None, all columns are used.

xscen.catalog.**ID_COLUMNS = ['bias_adjust_project', 'mip_era', 'activity', 'driving_model', 'institution', 'source', 'experiment', 'member', 'domain']**

Default columns used to create a unique ID

class xscen.catalog.**ProjectCatalog**(**args*, ***kwargs*)

Bases: *DataCatalog*

A DataCatalog with additional 'write' functionalities that can update and upload itself.

See also:

`intake_esm.core.esm_datastore`

classmethod create(*filename: PathLike | str*, ***, *project: dict | None = None*, *overwrite: bool = False*)

Create a new project catalog from some project metadata.

Creates the json from default `esm_col_data` and an empty csv file.

Parameters

- **filename** (*os.PathLike or str*) – A path to the json file (with or without suffix).
- **project** (*dict, optional*) – Metadata to create the catalog. If None, *CONFIG['project']* will be used. Valid fields are:
  - title : Name of the project, given as the catalog's "title".
  - **id**
    [slug-like version of the name, given as the catalog's id (should be url-proof)] Defaults to a modified name.

- version : Version of the project (and thus the catalog), string like "x.y.z".
- description : Detailed description of the project, given to the catalog's "description".
- Any other entry defined in `esm_col_data`.

At least one of *id* and *title* must be given, the rest is optional.

- **overwrite** (*bool*) – If True, will overwrite any existing JSON and CSV file.

Returns
ProjectCatalog – An empty intake_esm catalog.

`refresh()`

Reread the catalog CSV saved on disk.

`update(df: DataCatalog | esm_datastore | DataFrame | Series | Sequence[Series] | None = None)`

Update the catalog with new data and writes the new data to the csv file.

Once the internal dataframe is updated with *df*, the csv on disk is parsed, updated with the internal dataframe, duplicates are dropped and everything is written back to the csv. This means that nothing is _removed_* from the csv when calling this method, and it is safe to use even with a subset of the catalog.

Warning: If a file was deleted between the parsing of the catalog and this call, it will be removed from the csv when *check_valid* is called.

Parameters
df (*Union[DataCatalog, intake_esm.esm_datastore, pd.DataFrame, pd.Series, Sequence[pd.Series]], optional*) – Data to be added to the catalog. If None, nothing is added, but the catalog is still updated.

`update_from_ds(ds: Dataset, path: PathLike | str, info_dict: dict | None = None, **info_kwargs)`

Update the catalog with new data and writes the new data to the csv file.

We get the new data from the attributes of *ds*, the dictionary *info_dict* and *path*.

Once the internal dataframe is updated with the new data, the csv on disk is parsed, updated with the internal dataframe, duplicates are dropped and everything is written back to the csv. This means that nothing is _removed_* from the csv when calling this method, and it is safe to use even with a subset of the catalog.

Warning: If a file was deleted between the parsing of the catalog and this call, it will be removed from the csv when *check_valid* is called.

Parameters

- **ds** (*xarray.Dataset*) – Dataset that we want to add to the catalog. The columns of the catalog will be filled from the global attributes starting with 'cat:' of the dataset.
- **info_dict** (*dict, optional*) – Extra information to fill in the catalog.
- **path** (*os.PathLike or str*) – Path to the file that contains the dataset. This will be added to the 'path' column of the catalog.

xscen.catalog.**concat_data_catalogs**(**dcs*)

Concatenate a multiple DataCatalogs.

Output catalog is the union of all rows and all derived variables, with the "esmcat" of the first DataCatalog. Duplicate rows are dropped and the index is reset.

xscen.catalog.**generate_id**(*df: DataFrame | Dataset*, *id_columns: list | None = None*) → Series

Create an ID from column entries.

Parameters

- **df** (*pd.DataFrame, xr.Dataset*) – Data for which to create an ID.
- **id_columns** (*list, optional*) – List of column names on which to base the dataset definition. Empty columns will be skipped. If None (default), uses *ID_COLUMNS*.

Returns

pd.Series – A series of IDs, one per row of the input DataFrame.

xscen.catalog.**unstack_id**(*df: DataFrame | ProjectCatalog | DataCatalog*) → dict

Reverse-engineer an ID using catalog entries.

Parameters

df (*Union[pd.DataFrame, ProjectCatalog, DataCatalog]*) – Either a Project/DataCatalog or a pandas DataFrame.

Returns

dict – Dictionary with one entry per unique ID, which are themselves dictionaries of all the individual parts of the ID.

xscen.catutils module

Catalog creation and path building tools.

xscen.catutils.**build_path**(*data: dict | Dataset | DataArray | Series | DataCatalog | DataFrame*, *schemas: str | PathLike | dict | None = None*, *root: str | PathLike | None = None*, ***extra_facets*) → Path | *DataCatalog* | DataFrame

Parse the schema from a configuration and construct path using a dictionary of facets.

Parameters

- **data** (*dict or xr.Dataset or xr.DataArray or pd.Series or DataCatalog or pd.DataFrame*) – Dict of facets. Or xarray object to read the facets from. In the latter case, variable and time-dependent facets are read with *parse_from_ds()* and supplemented with all the object's attribute, giving priority to the "official" xscen attributes (prefixed with *cat:*, see *xscen.utils.get_cat_attrs()*). Can also be a catalog or a DataFrame, in which a "new_path" column is generated for each item.
- **schemas** (*Path or dict, optional*) – Path to YAML schematic of database schema. If None, will use a default schema. See the comments in the *xscen/data/file_schema.yml* file for more details on its construction. A dict of dict schemas can be given (same as reading the yaml). Or a single schema dict (single element of the yaml).
- **root** (*str or Path, optional*) – If given, the generated path(s) is given under this root one.
- ****extra_facets** – Extra facets to supplement or override metadadata missing from the first input.

Returns

Path or catalog – Constructed path. If "format" is absent from the facets, it has no suffix. If

data was a catalog, a copy with a "new_path" column is returned. Another "new_path_type" column is also added if *schemas* was a collection of schemas (like the default).

Examples

To rename a full catalog, the simplest way is to do:

```
>>> import xscen as xs
>>> import shutil as sh
>>> new_cat = xs.catutils.build_path(old_cat)
>>> for i, row in new_cat.iterrows():
...     sh.move(row.path, row.new_path)
...
```

xscen.catutils.**parse_directory**(*directories: list[str | PathLike]*, *patterns: list[str]*, ***, *id_columns: list[str] | None = None*, *read_from_file: bool | Sequence[str] | tuple[Sequence[str], Sequence[str]] | Sequence[tuple[Sequence[str], Sequence[str]]] = False*, *homogenous_info: dict | None = None*, *cvs: str | PathLike | dict | None = None*, *dirglob: str | None = None*, *xr_open_kwargs: Mapping[str, Any] | None = None*, *only_official_columns: bool = True*, *progress: bool = False*, *parallel_dirs: bool | int = False*, *file_checks: list[str] | None = None*) → DataFrame

Parse files in a directory and return them as a pd.DataFrame.

Parameters

- **directories** (*list of os.PathLike or list of str*) – List of directories to parse. The parse is recursive.
- **patterns** (*list of str*) – List of possible patterns to be used by `parse.parse()` to decode the file names. See Notes below.
- **id_columns** (*list of str, optional*) – List of column names on which to base the dataset definition. Empty columns will be skipped. If None (default), it uses `ID_COLUMNS`.
- **read_from_file** (*boolean or set of strings or tuple of 2 sets of strings or list of tuples*) – If True, if some fields were not parsed from their path, files are opened and missing fields are parsed from their metadata, if found. If a sequence of column names, only those fields are parsed from the file, if missing. If False (default), files are never opened. If a tuple of 2 lists of strings, only the first file of groups defined by the first list of columns is read and the second list of columns is parsed from the file and applied to the whole group. For example, *(["source"],["institution", "activity"])* will find a group with all the files that have the same source, open only one of the files to read the institution and activity, and write this information in the catalog for all filles of the group. It can also be a list of those tuples.
- **homogenous_info** (*dict, optional*) – Using the {column_name: description} format, information to apply to all files. These are applied before the *cvs*.
- **cvs** (*str or os.PathLike or dict, optional*) – Dictionary with mapping from parsed term to preferred terms (Controlled VocabularieS) for each column. May have an additional "attributes" entry which maps from attribute names in the files to official column names. The attribute translation is done before the rest. In the "variable" entry, if a name is mapped to None (null), that variable will not be listed in the catalog. A term can map to another mapping from field name to values, so that a value on one column triggers the filling of other columns. In the latter case, that other column must exist beforehand, whether it was in the pattern or in the homogenous_info.

- **dirglob** (*str, optional*) – A glob pattern for path matching to accelerate the parsing of a directory tree if only a subtree is needed. Only folders matching the pattern are parsed to find datasets.
- **xr_open_kwargs** (*dict*) – If needed, arguments to send xr.open_dataset() when opening the file to read the attributes.
- **only_official_columns** (*bool*) – If True (default), this ensures the final catalog only has the columns defined in *xscen.catalog.COLUMNS*. Other fields in the patterns will raise an error. If False, the columns are those used in the patterns and the homogenous info. In that case, the column order is not determined. Path, format and id are always present in the output.
- **progress** (*bool*) – If True, a counter is shown in stdout when finding files on disk. Does nothing if *parallel_dirs* is not False.
- **parallel_dirs** (*bool or int*) – If True, each directory is searched in parallel. If an int, it is the number of parallel searches. This should only be significantly useful if the directories are on different disks.
- **file_checks** (*list of str, optional*) – A list of file checks to run on the parsed files. Available values are: - "readable" : Check that the file is readable by the current user. - "writable" : Check that the file is writable by the current user. - "ncvalid" : For netCDF, check that it is valid (openable with netCDF4). Any check will slow down the parsing.

Notes

- **Offical columns names are controlled and ordered by `COLUMNS`:**

  ["id", "type", "processing_level", "mip_era", "activity", "driving_institution", "driving_model", "institution",
  "source", "bias_adjust_institution", "bias_adjust_project","experiment", "member", "xrfreq", "frequency", "variable", "domain", "date_start", "date_end", "version"]

- **Not all column names have to be present, but "xrfreq" (obtainable through "frequency"), "variable",**
  "date_start" and "processing_level" are necessary for a workable catalog.

- **'patterns' should highlight the columns with braces.**
  This acts like the reverse operation of *format()*. It is a template string with *{field name:type}* elements. The default "type" will match alphanumeric parts of the path, excluding the "_", "/" and "" characters. The "_" type will allow underscores. Field names prefixed by "?" will not be included in the output. See the documentation of `parse` for more type options. You can also add your own types using the *register_parse_type()* decorator.

  The "DATES" field is special as it will only match dates, either as a single date (YYYY, YYYYMM, YYYYMMDD) assigned to "{date_start}" (with "date_end" automatically inferred) or two dates of the same format as "{date_start}-{date_end}".

  Example: *"{source}/{?ignored project name}_{?:_}_{DATES}.nc"* Here, "source" will be the full folder name and it can't include underscores. The first section of the filename will be excluded from the output, it was given a name (ignore project name) to make the pattern readable. The last section of the filenames ("dates") will yield a "date_start" / "date_end" couple. All other sections in the middle will be ignored, as they match "{?:_}".

Returns
pd.DataFrame – Parsed directory files

xscen.catutils.**parse_from_ds**(*obj: str | PathLike | Dataset*, *names: Sequence[str]*, *attrs_map: Mapping[str, str] | None = None*, ***xrkwargs*)

Parse a list of catalog fields from the file/dataset itself.

If passed a path, this opens the file.

Infers the variable from the variables. Infers xrfreq, frequency, date_start and date_end from the time coordinate if present. Infers other attributes from the coordinates or the global attributes. Attributes names can be translated using the *attrs_map* mapping (from file attribute name to name in *names*).

If the obj is the path to a Zarr dataset and none of "frequency", "xrfreq", "date_start" or "date_end" are requested, `parse_from_zarr()` is used instead of opening the file.

Parameters

- **obj** (*str or os.PathLike or xr.Dataset*) – Dataset to parse.
- **names** (*sequence of str*) – List of attributes to be parsed from the dataset.
- **attrs_map** (*dict, optional*) – In the case of non-standard names in the file, this can be used to match entries in the files to specific 'names' in the requested list.
- **xrkwargs** – Arguments to be passed to open_dataset().

xscen.catutils.**register_parse_type**(*name: str*, *regex: str = '([^_\\/\\\\]*)'*, *group_count: int = 1*)

Register a new parse type to be available in *parse_directory()* patterns.

Function decorated by this will be registered in `EXTRA_PARSE_TYPES`. The function must take a single string and should return a single string. If you return a different type, it may interfere with the other steps of *parse_directory*.

Parameters

- **name** (*str*) – The type name. To make use of this type, put "{field:name}" in your pattern.
- **regex** (*str*) – A regex string to determine what can be matched by this type. The default matches anything but / and _, same as the default parse type.
- **group_count** (*int*) – The number of regex groups in the previous regex string.

xscen.config module

Configuration module.

Configuration in this module is taken from yaml files.

Functions wrapped by *parse_config()* have their kwargs automatically patched by values in the config.

The `CONFIG` dictionary contains all values, structured by submodules and functions. For example, for function `function` defined in `module.py` of this package, the config would look like:

```
module:
    function:
        ...kwargs...
```

The *load_config()* function fills the `CONFIG` dict from yaml files. It always updates the dictionary, so the latest file read has the highest priority.

At calling time, the priority order is always (from highest to lowest priority):

1. Explicitly passed keyword-args
2. Values in the loaded config
3. Function's default values.

Special sections

After parsing the files, *load_config()* will look into the config and perform some extra actions when finding the following special sections:

- `logging`: The content of this section will be sent directly to `logging.config.dictConfig()`.
- `xarray`: The content of this section will be sent directly to `xarray.set_options()`.
- `xclim`: The content of this section will be sent directly to `xclim.set_options()`. Here goes *metadata_locales: - fr* to activate the automatic translation of added attributes, for example.
- `warnings`: The content of this section must be a simple mapping. The keys are understood as python warning categories (types) and the values as an action to add to the filter. The key "all" applies the filter to any warnings. Only built-in warnings are supported.

xscen.config.**args_as_str**(**args: tuple[Any, ...]*) → tuple[str, ...]

Return arguments as strings.

xscen.config.**load_config**(**elements*, *reset: bool = False*, *verbose: bool = False*)

Load configuration from given files or key=value pairs.

Once all elements are loaded, special sections are dispatched to their module, but only if the section was changed by the loaded elements. These special sections are:

- *locales* : The locales to use when writing metadata in xscen, xclim and figanos. This section must be a list of 2-char strings.
- *logging* : Everything passed to `logging.config.dictConfig()`.
- *xarray* : Passed to `xarray.set_options()`.
- *xclim* : Passed to `xclim.set_options()`.
- *warning* : Mappings where the key is a Warning category (or "all") and the value an action to pass to `warnings.simplefilter()`.

Parameters

- **elements** (*str*) – Files or values to add into the config. If a directory is passed, all *.yml* files of this directory are added, in alphabetical order. If a "key=value" string, "key" is a dotted name and value will be evaluated if possible. "key=value" pairs are set last, after all files are being processed.
- **reset** (*bool*) – If True, the current config is erased before loading files.
- **verbose** (*bool*) – if True, each element triggers a INFO log line.

Example

```
load_config("my_config.yml", "config_dir/", "logging.loggers.xscen.level=DEBUG")
```

Will load configuration from *my_config.yml*, then from all yml files in *config_dir* and then the logging level of xscen's logger will be set to DEBUG.

xscen.config.**parse_config**(*func_or_cls*)

xscen.config.**recursive_update**(*d*, *other*)

Update a dictionary recursively with another dictionary.

Values that are Mappings are updated recursively as well.

xscen.diagnostics module

Functions to perform diagnostics on datasets.

xscen.diagnostics.**health_checks**(*ds: Dataset | DataArray*, ***, *structure: dict | None = None*, *calendar: str | None = None*, *start_date: str | None = None*, *end_date: str | None = None*, *variables_and_units: dict | None = None*, *cfchecks: dict | None = None*, *freq: str | None = None*, *missing: dict | str | list | None = None*, *flags: dict | None = None*, *flags_kwargs: dict | None = None*, *return_flags: bool = False*, *raise_on: list | None = None*) → None | Dataset

Perform a series of health checks on the dataset. Be aware that missing data checks and flag checks can be slow.

Parameters

- **ds** (*xr.Dataset or xr.DataArray*) – Dataset to check.
- **structure** (*dict, optional*) – Dictionary with keys "dims" and "coords" containing the expected dimensions and coordinates. This check will fail is extra dimensions or coordinates are found.
- **calendar** (*str, optional*) – Expected calendar. Synonyms should be detected correctly (e.g. "standard" and "gregorian").
- **start_date** (*str, optional*) – To check if the dataset starts at least at this date.
- **end_date** (*str, optional*) – To check if the dataset ends at least at this date.
- **variables_and_units** (*dict, optional*) – Dictionary containing the expected variables and units.
- **cfchecks** (*dict, optional*) – Dictionary where the key is the variable to check and the values are the cfchecks. The cfchecks themselves must be a dictionary with the keys being the cfcheck names and the values being the arguments to pass to the cfcheck. See *xclim.core.cfchecks* for more details.
- **freq** (*str, optional*) – Expected frequency, written as the result of xr.infer_freq(ds.time).
- **missing** (*dict or str or list of str, optional*) – String, list of strings, or dictionary where the key is the method to check for missing data and the values are the arguments to pass to the method. The methods are: "missing_any", "at_least_n_valid", "missing_pct", "missing_wmo". See `xclim.core.missing()` for more details.
- **flags** (*dict, optional*) – Dictionary where the key is the variable to check and the values are the flags. The flags themselves must be a dictionary with the keys being the data_flags names and the values being the arguments to pass to the data_flags. If *None* is passed instead of a dictionary, then xclim's default flags for the given variable are run. See `xclim.core.utils.VARIABLES`. See also `xclim.core.dataflags.data_flags()` for the list of possible flags.
- **flags_kwargs** (*dict, optional*) – Additional keyword arguments to pass to the data_flags ("dims" and "freq").
- **return_flags** (*bool*) – Whether to return the Dataset created by data_flags.
- **raise_on** (*list of str, optional*) – Whether to raise an error if a check fails, else there will only be a warning. The possible values are the names of the checks. Use ["all"] to raise on all checks.

Returns

xr.Dataset or None – Dataset containing the flags if return_flags is True & raise_on is False for the "flags" check.

xscen.diagnostics.**measures_heatmap**(*meas_datasets: list[Dataset] | dict*, *to_level: str = 'diag-heatmap'*) → Dataset

Create a heatmap to compare the performance of the different datasets.

The columns are properties and the rows are datasets. Each point is the absolute value of the mean of the measure over the whole domain. Each column is normalized from 0 (best) to 1 (worst).

Parameters

- **meas_datasets** (*list of xr.Dataset or dict*) – List or dictionary of datasets of measures of properties. If it is a dictionary, the keys will be used to name the rows. If it is a list, the rows will be given a number.
- **to_level** (*str*) – The processing_level to assign to the output.

Returns

xr.Dataset – Dataset containing the heatmap.

xscen.diagnostics.**measures_improvement**(*meas_datasets: list[Dataset] | dict*, *to_level: str = 'diag-improved'*) → Dataset

Calculate the fraction of improved grid points for each property between two datasets of measures.

Parameters

- **meas_datasets** (*list of xr.Dataset or dict*) – List of 2 datasets: Initial dataset of measures and final (improved) dataset of measures. Both datasets must have the same variables. It is also possible to pass a dictionary where the values are the datasets and the key are not used.
- **to_level** (*str*) – processing_level to assign to the output dataset

Returns

xr.Dataset – Dataset containing information on the fraction of improved grid points for each property.

xscen.diagnostics.**properties_and_measures**(*ds: Dataset*, *properties: str | PathLike | Sequence[Indicator] | Sequence[tuple[str, Indicator]] | module*, *period: list[str] | None = None*, *unstack: bool = False*, *rechunk: dict | None = None*, *dref_for_measure: Dataset | None = None*, *change_units_arg: dict | None = None*, *to_level_prop: str = 'diag-properties'*, *to_level_meas: str = 'diag-measures'*) → tuple[Dataset, Dataset]

Calculate properties and measures of a dataset.

Parameters

- **ds** (*xr.Dataset*) – Input dataset.
- **properties** (*Union[str, os.PathLike, Sequence[Indicator], Sequence[tuple[str, Indicator]], ModuleType]*) – Path to a YAML file that instructs on how to calculate properties. Can be the indicator module directly, or a sequence of indicators or a sequence of tuples (indicator name, indicator) as returned by *iter_indicators()*.
- **period** (*list of str, optional*) – [start, end] of the period to be evaluated. The period will be selected on ds and dref_for_measure if it is given.
- **unstack** (*bool*) – Whether to unstack ds before computing the properties.
- **rechunk** (*dict, optional*) – Dictionary of chunks to use for a rechunk before computing the properties.

- **dref_for_measure** (*xr.Dataset, optional*) – Dataset of properties to be used as the ref argument in the computation of the measure. Ideally, this is the first output (prop) of a previous call to this function. Only measures on properties that are provided both in this dataset and in the properties list will be computed. If None, the second output of the function (meas) will be an empty Dataset.
- **change_units_arg** (*dict, optional*) – If not None, calls *xscen.utils.change_units* on ds before computing properties using this dictionary for the *variables_and_units* argument. It can be useful to convert units before computing the properties, because it is sometimes easier to convert the units of the variables than the units of the properties (e.g. variance).
- **to_level_prop** (*str*) – processing_level to give the first output (prop)
- **to_level_meas** (*str*) – processing_level to give the second output (meas)

Returns

- **prop** (*xr.Dataset*) – Dataset of properties of ds
- **meas** (*xr.Dataset*) – Dataset of measures between prop and dref_for_meas

See also:

`xclim.sdba.properties`, `xclim.sdba.measures`, `xclim.core.indicator.build_indicator_module_from_yaml`

xscen.ensembles module

Ensemble statistics and weights.

`xscen.ensembles.`**`build_partition_data`**(*datasets: dict | list[Dataset]*, *partition_dim: list[str] = ['source', 'experiment', 'bias_adjust_project']*, *subset_kw: dict = None*, *regrid_kw: dict = None*, *rename_dict: dict = None*)

Get the input for the xclim partition functions.

From a list or dictionary of datasets, create a single dataset with *partition_dim* dimensions (and time) to pass to one of the xclim partition functions (https://xclim.readthedocs.io/en/stable/api.html#uncertainty-partitioning). If the inputs have different grids, they have to be subsetted and regridded to a common grid/point.

Parameters

- **datasets** (*dict*) – List or dictionnary of Dataset objects that will be included in the ensemble. The datasets should include the necessary ("cat:") attributes to understand their metadata. Tip: With a project catalog, you can do: *datasets = pcat.search(**search_dict).to_dataset_dict()*.
- **partition_dim** (*list[str]*) – Components of the partition. They will become the dimension of the output. The default is ['source', 'experiment', 'bias_adjust_project']. For source, the dimension will actually be institution_source_member.
- **subset_kw** (*dict*) – Arguments to pass to *xs.spatial.subset()*.
- **regrid_kw** – Arguments to pass to *xs.regrid_dataset()*.
- **rename_dict** – Dictionary to rename the dimensions from xscen names to xclim names. The default is {'source': 'model', 'bias_adjust_project': 'downscaling', 'experiment': 'scenario'}.

Returns

xr.Dataset – The input data for the partition functions.

See also:

xclim.ensembles

xscen.ensembles.**ensemble_stats**(*datasets: dict | list[str | PathLike] | list[Dataset] | list[DataArray] | Dataset*, *statistics: dict*, ***, *create_kwargs: dict | None = None*, *weights: DataArray | None = None*, *common_attrs_only: bool = True*, *to_level: str = 'ensemble'*) → Dataset

Create an ensemble and computes statistics on it.

Parameters

- **datasets** (*dict or list of [str, os.PathLike, Dataset or DataArray], or Dataset*) – List of file paths or xarray Dataset/DataArray objects to include in the ensemble. A dictionary can be passed instead of a list, in which case the keys are used as coordinates along the new *realization* axis. Tip: With a project catalog, you can do: *datasets = pcat.search(**search_dict).to_dataset_dict()*. If a single Dataset is passed, it is assumed to already be an ensemble and will be used as is. The 'realization' dimension is required.
- **statistics** (*dict*) – xclim.ensembles statistics to be called. Dictionary in the format {function: arguments}. If a function requires 'weights', you can leave it out of this dictionary and it will be applied automatically if the 'weights' argument is provided. See the Notes section for more details on robustness statistics, which are more complex in their usage.
- **create_kwargs** (*dict, optional*) – Dictionary of arguments for xclim.ensembles.create_ensemble.
- **weights** (*xr.DataArray, optional*) – Weights to apply along the 'realization' dimension. This array cannot contain missing values.
- **common_attrs_only** (*bool*) – If True, keeps only the global attributes that are the same for all datasets and generate new id. If False, keeps global attrs of the first dataset (same behaviour as xclim.ensembles.create_ensemble)
- **to_level** (*str*) – The processing level to assign to the output.

Returns

xr.Dataset – Dataset with ensemble statistics

Notes

- The positive fraction in 'change_significance' and 'robustness_fractions' is calculated by xclim using 'v > 0', which is not appropriate for relative deltas. This function will attempt to detect relative deltas by using the 'delta_kind' attribute ('rel.', 'relative', '*', or '/') and will apply 'v - 1' before calling the function.
- The 'robustness_categories' statistic requires the outputs of 'robustness_fractions'. Thus, there are two ways to build the 'statistics' dictionary:
  1. Having 'robustness_fractions' and 'robustness_categories' as separate entries in the dictionary. In this case, all outputs will be returned.
  2. Having 'robustness_fractions' as a nested dictionary under 'robustness_categories'. In this case, only the robustness categories will be returned.
- A 'ref' DataArray can be passed to 'change_significance' and 'robustness_fractions', which will be used by xclim to compute deltas and perform some significance tests. However, this supposes that both 'datasets' and 'ref' are still timeseries (e.g. annual means), not climatologies where the 'time' dimension represents the period over which the climatology was computed. Thus, using 'ref' is only accepted if 'robustness_fractions' (or 'robustness_categories') is the only statistic being computed.

- If you want to use compute a robustness statistic on a climatology, you should first compute the climatologies and deltas yourself, then leave 'ref' as None and pass the deltas as the 'datasets' argument. This will be compatible with other statistics.

See also:

xclim.ensembles._base.create_ensemble, xclim.ensembles._base.ensemble_percentiles, xclim.ensembles._base.ensemble_mean_std_max_min, xclim.ensembles._robustness.robustness_fractions, xclim.ensembles._robustness.robustness_categories, xclim.ensembles._robustness.robustness_coefficient

xscen.ensembles.**generate_weights**(*datasets: dict | list*, ***, *independence_level: str = 'model'*, *balance_experiments: bool = False*, *attribute_weights: dict | None = None*, *skipna: bool = True*, *v_for_skipna: str | None = None*, *standardize: bool = False*, *experiment_weights: bool = False*) → DataArray

Use realization attributes to automatically generate weights along the 'realization' dimension.

Parameters

- **datasets** (*dict*) – List of Dataset objects that will be included in the ensemble. The datasets should include the necessary attributes to understand their metadata - See 'Notes' below. A dictionary can be passed instead of a list, in which case the keys are used for the 'realization' coordinate. Tip: With a project catalog, you can do: *datasets = pcat.search(**search_dict).to_dataset_dict()*.
- **independence_level** (*str*) – 'model': Weights using the method '1 model - 1 Vote', where every unique combination of 'source' and 'driving_model' is considered a model. 'GCM': Weights using the method '1 GCM - 1 Vote' 'institution': Weights using the method '1 institution - 1 Vote'
- **balance_experiments** (*bool*) – If True, each experiment will be given a total weight of 1 (prior to subsequent weighting made through *attribute_weights*). This option requires the 'cat:experiment' attribute to be present in all datasets.
- **attribute_weights** (*dict, optional*) – Nested dictionaries of weights to apply to each dataset. These weights are applied after the independence weighting. The first level of keys are the attributes for which weights are being given. The second level of keys are unique entries for the attribute, with the value being either an individual weight or a xr.DataArray. If a DataArray is used, its dimensions must be the same non-stationary coordinate as the datasets (ex: time, horizon) and the attribute being weighted (ex: experiment). A *others* key can be used to give the same weight to all entries not specifically named in the dictionary. Example #1: {'source': {'MPI-ESM-1-2-HAM': 0.25, 'MPI-ESM1-2-HR': 0.5}}, Example #2: {'experiment': {'ssp585': xr.DataArray, 'ssp126': xr.DataArray}, 'institution': {'CCCma': 0.5, 'others': 1}}
- **skipna** (*bool*) – If True, weights will be computed from attributes only. If False, weights will be computed from the number of non-missing values. skipna=False requires either a 'time' or 'horizon' dimension in the datasets.
- **v_for_skipna** (*str, optional*) – Variable to use for skipna=False. If None, the first variable in the first dataset is used.
- **standardize** (*bool*) – If True, the weights are standardized to sum to 1 (per timestep/horizon, if skipna=False).
- **experiment_weights** (*bool*) – Deprecated. Use balance_experiments instead.

Notes

The following attributes are required for the function to work:

- 'cat:source' in all datasets
- 'cat:driving_model' in regional climate models
- 'cat:institution' in all datasets if independence_level='institution'
- 'cat:experiment' in all datasets if split_experiments=True

Even when not required, the 'cat:member' and 'cat:experiment' attributes are strongly recommended to ensure the weights are computed correctly.

Returns

xr.DataArray – Weights along the 'realization' dimension, or 2D weights along the 'realization' and 'time/horizon' dimensions if skipna=False.

xscen.extract module

Functions to find and extract data from a catalog.

xscen.extract.**extract_dataset**(*catalog:* DataCatalog, *, *variables_and_freqs: dict | None = None*, *periods: list[str] | list[list[str]] | None = None*, *region: dict | None = None*, *to_level: str = 'extracted'*, *ensure_correct_time: bool = True*, *xr_open_kwargs: dict | None = None*, *xr_combine_kwargs: dict | None = None*, *preprocess: Callable | None = None*, *resample_methods: dict | None = None*, *mask: bool | Dataset | DataArray = False*) → dict

Take one element of the output of *search_data_catalogs* and returns a dataset, performing conversions and resampling as needed.

Nothing is written to disk within this function.

Parameters

- **catalog** (*DataCatalog*) – Sub-catalog for a single dataset, one value of the output of *search_data_catalogs*.
- **variables_and_freqs** (*dict, optional*) – Variables and freqs, following a 'variable: xrfreq-compatible str' format. A list of strings can also be provided. If None, it will be read from catalog._requested_variables and catalog._requested_variable_freqs (set by *variables_and_freqs* in *search_data_catalogs*)
- **periods** (*list of str or list of lists of str, optional*) – Either [start, end] or list of [start, end] for the periods to be evaluated. Will be read from catalog._requested_periods if None. Leave both None to extract everything.
- **region** (*dict, optional*) – Description of the region and the subsetting method (required fields listed in the Notes) used in *xscen.spatial.subset*.
- **to_level** (*str*) – The processing level to assign to the output. Defaults to 'extracted'
- **ensure_correct_time** (*bool*) – When True (default), even if the data has the correct frequency, its time coordinate is checked so that it exactly matches the frequency code (xrfreq). For example, daily data given at noon would be transformed to be given at midnight. If the time coordinate is invalid, it raises an error.
- **xr_open_kwargs** (*dict, optional*) – A dictionary of keyword arguments to pass to *DataCatalogs.to_dataset_dict*, which will be passed to *xr.open_dataset*.

- **xr_combine_kwargs** (*dict, optional*) – A dictionary of keyword arguments to pass to *DataCatalogs.to_dataset_dict*, which will be passed to *xr.combine_by_coords*.
- **preprocess** (*callable, optional*) – If provided, call this function on each dataset prior to aggregation.
- **resample_methods** (*dict, optional*) – Dictionary where the keys are the variables and the values are the resampling method. Options for the resampling method are {'mean', 'min', 'max', 'sum', 'wind_direction'}. If the method is not given for a variable, it is guessed from the variable name and frequency, using the mapping in CVs/resampling_methods.json. If the variable is not found there, "mean" is used by default.
- **mask** (*xr.Dataset or xr.DataArray or bool*) – A mask that is applied to all variables and only keeps data where it is True. Where the mask is False, variable values are replaced by NaNs. The mask should have the same dimensions as the variables extracted. If *mask* is a dataset, the dataset should have a variable named 'mask'. If *mask* is True, it will expect a *mask* variable at xrfreq *fx* to have been extracted.

Returns

dict – Dictionary (keys = xrfreq) with datasets containing all available and computed variables, subsetted to the region, everything resampled to the requested frequency.

Notes

'region' fields:

name: str
Region name used to overwrite domain in the catalog.

method: str
['gridpoint', 'bbox', shape', 'sel']

tile_buffer: float, optional
Multiplier to apply to the model resolution.

kwargs
Arguments specific to the method used.

See also:

`intake_esm.core.esm_datastore.to_dataset_dict`, `xarray.open_dataset`, `xarray.combine_by_coords`

`xscen.extract.`**`get_warming_level`**(*realization: Dataset | DataArray | dict | Series | DataFrame | str | list*, *wl: float*, ***, *window: int = 20*, *tas_baseline_period: Sequence[str] | None = None*, *ignore_member: bool = False*, *tas_src: str | PathLike | None = None*, *return_horizon: bool = True*) → dict | list[str] | str

Use the IPCC Atlas method to return the window of time over which the requested level of global warming is first reached.

Parameters

- **realization** (*xr.Dataset, xr.DataArray, dict, str, Series or sequence of those*) – Model to be evaluated. Needs the four fields mip_era, source, experiment and member, as a dict or in a Dataset's attributes. Strings should follow this formatting: {mip_era}_{source}_{experiment}_{member}. Lists of dicts, strings or Datasets are also accepted, in which case the output will be a dict. Regex wildcards (.*) are accepted, but may lead to unexpected results. Datasets should include the catalogue attributes (starting

by "cat:") required to create such a string: 'cat:mip_era', 'cat:experiment', 'cat:member', and either 'cat:source' for global models or 'cat:driving_model' for regional models. e.g. 'CMIP5_CanESM2_rcp85_r1i1p1'

- **wl** (*float*) – Warming level. e.g. 2 for a global warming level of +2 degree Celsius above the mean temperature of the *tas_baseline_period*.
- **window** (*int*) – Size of the rolling window in years over which to compute the warming level.
- **tas_baseline_period** (*list, optional*) – [start, end] of the base period. The warming is calculated with respect to it. The default is ["1850", "1900"].
- **ignore_member** (*bool*) – Decides whether to ignore the member when searching for the model run in tas_csv.
- **tas_src** (*str, optional*) – Path to a netCDF of annual global mean temperature (tas) with an annual "time" dimension and a "simulation" dimension with the following coordinates: "mip_era", "source", "experiment" and "member". If None, it will default to data/IPCC_annual_global_tas.nc which was built from the IPCC atlas data from Iturbide et al., 2020 (https://doi.org/10.5194/essd-12-2959-2020) and extra data for missing CMIP6 models and pilot models of CRCM5 and ClimEx.
- **return_horizon** (*bool*) – If True, the output will be a list following the format ['start_yr', 'end_yr'] If False, the output will be a string representing the middle of the period.

Returns

dict, list or str – If *realization* is not a sequence, the output will follow the format indicated by *return_horizon*. If *realization* is a sequence, the output will be a list or dictionary depending on *output*, with values following the format indicated by *return_horizon*.

xscen.extract.**resample**(*da: DataArray*, *target_frequency: str*, ***, *ds: Dataset | None = None*, *method: str | None = None*, *missing: str | dict | None = None*) → DataArray

Aggregate variable to the target frequency.

If the input frequency is greater than a week, the resampling operation is weighted by the number of days in each sampling period.

Parameters

- **da** (*xr.DataArray*) – DataArray of the variable to resample, must have a "time" dimension and be of a finer temporal resolution than "target_frequency".
- **target_frequency** (*str*) – The target frequency/freq str, must be one of the frequency supported by xarray.
- **ds** (*xr.Dataset, optional*) – The "wind_direction" resampling method needs extra variables, which can be given here.
- **method** (*{'mean', 'min', 'max', 'sum', 'wind_direction'}, optional*) – The resampling method. If None (default), it is guessed from the variable name and frequency, using the mapping in CVs/resampling_methods.json. If the variable is not found there, "mean" is used by default.
- **missing** (*{'mask', 'drop'} or dict, optional*) – If 'mask' or 'drop', target periods that would have been computed from fewer timesteps than expected are masked or dropped, using a threshold of 5% of missing data. E.g. the first season of a *target_frequency* of "QS-DEC" will be masked or dropped if data starts in January. If a dict, points to a xclim check missing method which will mask periods according to the number of NaN values. The dict must contain a "method" field corresponding to the xclim method name and may contain any other args to pass. Options are documented in `xclim.core.missing`.

Returns

xr.DataArray – Resampled variable

xscen.extract.**search_data_catalogs**(*data_catalogs: str | PathLike | DataCatalog | list[str | PathLike | DataCatalog]*, *variables_and_freqs: dict*, ***, *other_search_criteria: dict | None = None*, *exclusions: dict | None = None*, *match_hist_and_fut: bool = False*, *periods: list[str] | list[list[str]] | None = None*, *coverage_kwargs: dict | None = None*, *id_columns: list[str] | None = None*, *allow_resampling: bool = False*, *allow_conversion: bool = False*, *conversion_yaml: str | None = None*, *restrict_resolution: str | None = None*, *restrict_members: dict | None = None*, *restrict_warming_level: dict | bool | None = None*) → dict

Search through DataCatalogs.

Parameters

- **data_catalogs** (*str, os.PathLike, DataCatalog, or a list of those*) – DataCatalog (or multiple, in a list) or paths to JSON/CSV data catalogs. They must use the same columns and aggregation options.
- **variables_and_freqs** (*dict*) – Variables and freqs to search for, following a 'variable: xr-freq-compatible-str' format. A list of strings can also be provided.
- **other_search_criteria** (*dict, optional*) – Other criteria to search for in the catalogs' columns, following a 'column_name: list(subset)' format. You can also pass 'require_all_on: list(columns_name)' in order to only return results that correspond to all other criteria across the listed columns. More details available at https://intake-esm.readthedocs.io/en/stable/how-to/enforce-search-query-criteria-via-require-all-on.html .
- **exclusions** (*dict, optional*) – Same as other_search_criteria, but for eliminating results. Any result that matches any of the exclusions will be removed.
- **match_hist_and_fut** (*bool*) – If True, historical and future simulations will be combined into the same line, and search results lacking one of them will be rejected.
- **periods** (*list of str or list of lists of str, optional*) – Either [start, end] or list of [start, end] for the periods to be evaluated.
- **coverage_kwargs** (*dict, optional*) – Arguments to pass to subset_file_coverage (only used when periods is not None).
- **id_columns** (*list, optional*) – List of columns used to create a id column. If None is given, the original "id" is left.
- **allow_resampling** (*bool*) – If True (default), variables with a higher time resolution than requested are considered.
- **allow_conversion** (*bool*) – If True (default) and if the requested variable cannot be found, intermediate variables are searched given that there exists a converting function in the "derived variable registry".
- **conversion_yaml** (*str, optional*) – Path to a YAML file that defines the possible conversions (used alongside 'allow_conversion'=True). This file should follow the xclim conventions for building a virtual module. If None, the "derived variable registry" will be defined by the file in "xscen/xclim_modules/conversions.yml"
- **restrict_resolution** (*str, optional*) – Used to restrict the results to the finest/coarsest resolution available for a given simulation. ['finest', 'coarsest'].
- **restrict_members** (*dict, optional*) – Used to restrict the results to a given number of members for a given simulation. Currently only supports {"ordered": int} format.

- **restrict_warming_level** (*bool or dict, optional*) – Used to restrict the results only to datasets that exist in the csv used to compute warming levels in *subset_warming_level*. If True, this will only keep the datasets that have a mip_era, source, experiment and member combination that exist in the csv. This does not guarantee that a given warming level will be reached, only that the datasets have corresponding columns in the csv. More option can be added by passing a dictionary instead of a boolean. If {'ignore_member':True}, it will disregard the member when trying to match the dataset to a column. If {tas_src: Path_to_netcdf}, it will use an alternative netcdf instead of the default one provided by xscen. If 'wl' is a provided key, then *xs.get_warming_level* will be called and only datasets that reach the given warming level will be kept. This can be combined with other arguments of the function, for example {'wl': 1.5, 'window': 30}.

Notes

- The "other_search_criteria" and "exclusions" arguments accept wildcard (*) and regular expressions.
- Frequency can be wildcarded with 'NA' in the *variables_and_freqs* dict.
- Variable names cannot be wildcarded, they must be CMIP6-standard.

Returns

dict – Keys are the id and values are the DataCatalogs for each entry. A single DataCatalog can be retrieved with *concat_data_catalogs(*out.values())*. Each DataCatalog has a subset of the derived variable registry that corresponds to the needs of this specific group. Usually, each entry can be written to file in a single Dataset when using *extract_dataset* with the same arguments.

See also:

`intake_esm.core.esm_datastore.search`

xscen.extract.**subset_warming_level**(*ds: Dataset*, *wl: float | Sequence[float]*, *to_level: str = 'warminglevel-{wl}vs{period0}-{period1}'*, *wl_dim: str | bool = '+{wl}Cvs{period0}-{period1}'*, ***kwargs*) → Dataset | None

Subsets the input dataset with only the window of time over which the requested level of global warming is first reached, using the IPCC Atlas method.

Parameters

- **ds** (*xr.Dataset*) – Input dataset. The dataset should include attributes to help recognize it and find its warming levels - 'cat:mip_era', 'cat:experiment', 'cat:member', and either 'cat:source' for global models or 'cat:driving_institution' (optional) + 'cat:driving_model' for regional models. Or , it should include a *realization* dimension constructed as "{mip_era}_{source or driving_model}_{experiment}_{member}" for vectorized subsetting. Vectorized subsetting is currently only implemented for annual data.
- **wl** (*float or sequence of floats*) – Warming level. e.g. 2 for a global warming level of +2 degree Celsius above the mean temperature of the *tas_baseline_period*. Multiple levels can be passed, in which case using "{wl}" in *to_level* and *wl_dim* is not recommended. Mutliple levels are currently only implemented for annual data.
- **to_level** – The processing level to assign to the output. Use "{wl}", "{period0}" and "{period1}" in the string to dynamically include *wl*, 'tas_baseline_period[0]' and 'tas_baseline_period[1]'.

- **wl_dim** (*str or boolean, optional*) – The value to use to fill the new *warminglevel* dimension. Use "{wl}", "{period0}" and "{period1}" in the string to dynamically include *wl*, 'tas_baseline_period[0]' and 'tas_baseline_period[1]'. If None, no new dimensions will be added, invalid if *wl* is a sequence. If True, the dimension will include *wl* as numbers and units of "degC".
- ****kwargs** – Instructions on how to search for warming levels, passed to `get_warming_level()`.

Returns

xr.Dataset or None – Warming level dataset, or None if *ds* can't be subsetted for the requested warming level. The dataset will have a new dimension *warminglevel* with *wl_dim* as coordinates. If *wl* was a list or if ds had a "realization" dim, the "time" axis is replaced by a fake time starting in 1000-01-01 and with a length of *window* years. Start and end years of the subsets are bound in the new coordinate "warminglevel_bounds".

xscen.indicators module

Functions to compute xclim indicators.

xscen.indicators.**compute_indicators**(*ds: Dataset*, *indicators: str | PathLike | Sequence[Indicator] | Sequence[tuple[str, Indicator]] | module*, ***, *periods: list[str] | list[list[str]] | None = None*, *restrict_years: bool = True*, *to_level: str | None = 'indicators'*) → dict

Calculate variables and indicators based on a YAML call to xclim.

The function cuts the output to be the same years as the inputs. Hence, if an indicator creates a timestep outside the original year range (e.g. the first DJF for QS-DEC), it will not appear in the output.

Parameters

- **ds** (*xr.Dataset*) – Dataset to use for the indicators.
- **indicators** (*Union[str, os.PathLike, Sequence[Indicator], Sequence[tuple[str, Indicator]], ModuleType]*) – Path to a YAML file that instructs on how to calculate missing variables. Can also be only the "stem", if translations and custom indices are implemented. Can be the indicator module directly, or a sequence of indicators or a sequence of tuples (indicator name, indicator) as returned by *iter_indicators()*.
- **periods** (*list of str or list of lists of str, optional*) – Either [start, end] or list of [start, end] of continuous periods over which to compute the indicators. This is needed when the time axis of ds contains some jumps in time. If None, the dataset will be considered continuous.
- **restrict_years** (*bool*) – If True, cut the time axis to be within the same years as the input. This is mostly useful for frequencies that do not start in January, such as QS-DEC. In that instance, *xclim* would start on previous_year-12-01 (DJF), with a NaN. *restrict_years* will cut that first timestep. This should have no effect on YS and MS indicators.
- **to_level** (*str, optional*) – The processing level to assign to the output. If None, the processing level of the inputs is preserved.

Returns

dict – Dictionary (keys = timedeltas) with indicators separated by temporal resolution.

See also:

`xclim.indicators`, `xclim.core.indicator.build_indicator_module_from_yaml`

xscen.indicators.**load_xclim_module**(*filename: str | PathLike*, *reload: bool = False*) → module

Return the xclim module described by the yaml file (or group of yaml, jsons and py).

Parameters

- **filename** (*str or os.PathLike*) – The filepath to the yaml file of the module or to the stem of yaml, jsons and py files.
- **reload** (*bool*) – If False (default) and the module already exists in *xclim.indicators*, it is not re-build.

Returns

ModuleType – The xclim module.

xscen.io module

Input/Output functions for xscen.

xscen.io.**clean_incomplete**(*path: str | PathLike*, *complete: Sequence[str]*) → None

Delete un-catalogued variables from a zarr folder.

The goal of this function is to clean up an incomplete calculation. It will remove any variable in the zarr that is neither in the *complete* list nor in the *coords*.

Parameters

- **path** (*str, Path*) – A path to a zarr folder.
- **complete** (*sequence of strings*) – Name of variables that were completed.

Returns

None

xscen.io.**estimate_chunks**(*ds: str | PathLike | Dataset*, *dims: list*, *target_mb: float = 50*, *chunk_per_variable: bool = False*) → dict

Return an approximate chunking for a file or dataset.

Parameters

- **ds** (*xr.Dataset, str*) – Either a xr.Dataset or the path to a NetCDF file. Existing chunks are not taken into account.
- **dims** (*list*) – Dimension(s) on which to estimate the chunking. Not implemented for more than 2 dimensions.
- **target_mb** (*float*) – Roughly the size of chunks (in Mb) to aim for.
- **chunk_per_variable** (*bool*) – If True, the output will be separated per variable. Otherwise, a common chunking will be found.

Returns

dict – A dictionary mapping dimensions to chunk sizes.

xscen.io.**get_engine**(*file: str | PathLike*) → str

Use functionality of h5py to determine if a NetCDF file is compatible with h5netcdf.

Parameters

file (*str or os.PathLike*) – Path to the file.

Returns

str – Engine to use with xarray

xscen.io.**make_toc**(*ds: Dataset | DataArray*, *loc: str | None = None*) → DataFrame

Make a table of content describing a dataset's variables.

This return a simple DataFrame with variable names as index, the long_name as "description" and units. Column names and long names are taken from the activated locale if found, otherwise the english version is taken.

Parameters

- **ds** (*xr.Dataset or xr.DataArray*) – Dataset or DataArray from which to extract the relevant metadata.
- **loc** (*str, optional*) – The locale to use. If None, either the first locale in the list of activated xclim locales is used, or "en" if none is activated.

Returns

pd.DataFrame – A DataFrame with variables as index, and columns "description" and "units".

xscen.io.**rechunk**(*path_in: PathLike | str | Dataset*, *path_out: PathLike | str*, ***, *chunks_over_var: dict | None = None*, *chunks_over_dim: dict | None = None*, *worker_mem: str*, *temp_store: str | PathLike | None = None*, *overwrite: bool = False*) → None

Rechunk a dataset into a new zarr.

Parameters

- **path_in** (*path, str or xr.Dataset*) – Input to rechunk.
- **path_out** (*path or str*) – Path to the target zarr.
- **chunks_over_var** (*dict*) – Mapping from variables to mappings from dimension name to size. Give this argument or *chunks_over_dim*.
- **chunks_over_dim** (*dict*) – Mapping from dimension name to size that will be used for all variables in ds. Give this argument or *chunks_over_var*.
- **worker_mem** (*str*) – The maximal memory usage of each task. When using a distributed Client, this an approximate memory per thread. Each worker of the client should have access to 10-20% more memory than this times the number of threads.
- **temp_store** (*path or str, optional*) – A path to a zarr where to store intermediate results.
- **overwrite** (*bool*) – If True, it will delete whatever is in path_out before doing the rechunking.

Returns

None

See also:

`rechunker.rechunk`

xscen.io.**rechunk_for_saving**(*ds: Dataset*, *rechunk: dict*)

Rechunk before saving to .zarr or .nc, generalized as Y/X for different axes lat/lon, rlat/rlon.

Parameters

- **ds** (*xr.Dataset*) – The xr.Dataset to be rechunked.
- **rechunk** (*dict*) – A dictionary with the dimension names of ds and the new chunk size. Spatial dimensions can be provided as X/Y.

Returns

xr.Dataset – The dataset with new chunking.

xscen.io.**round_bits**(*da: DataArray*, *keepbits: int*)

Round floating point variable by keeping a given number of bits in the mantissa, dropping the rest. This allows for a much better compression.

Parameters

- **da** (*xr.DataArray*) – Variable to be rounded.
- **keepbits** (*int*) – The number of bits of the mantissa to keep.

xscen.io.**save_to_netcdf**(*ds: Dataset*, *filename: str | PathLike*, ***, *rechunk: dict | None = None*, *bitround: bool | int | dict = False*, *compute: bool = True*, *netcdf_kwargs: dict | None = None*)

Save a Dataset to NetCDF, rechunking or compressing if requested.

Parameters

- **ds** (*xr.Dataset*) – Dataset to be saved.
- **filename** (*str or os.PathLike*) – Name of the NetCDF file to be saved.
- **rechunk** (*dict, optional*) – This is a mapping from dimension name to new chunks (in any format understood by dask). Spatial dimensions can be generalized as 'X' and 'Y', which will be mapped to the actual grid type's dimension names. Rechunking is only done on *data* variables sharing dimensions with this argument.
- **bitround** (*bool or int or dict*) – If not False, float variables are bit-rounded by dropping a certain number of bits from their mantissa, allowing for a much better compression. If an int, this is the number of bits to keep for all float variables. If a dict, a mapping from variable name to the number of bits to keep. If True, the number of bits to keep is guessed based on the variable's name, defaulting to 12, which yields a relative error below 0.013%.
- **compute** (*bool*) – Whether to start the computation or return a delayed object.
- **netcdf_kwargs** (*dict, optional*) – Additional arguments to send to_netcdf()

Returns

None

See also:

xarray.Dataset.to_netcdf

xscen.io.**save_to_table**(*ds: Dataset | DataArray*, *filename: str | PathLike*, *output_format: str | None = None*, ***, *row: str | Sequence[str] | None = None*, *column: None | str | Sequence[str] = 'variable'*, *sheet: str | Sequence[str] | None = None*, *coords: bool | Sequence[str] = True*, *col_sep: str = '_'*, *row_sep: str | None = None*, *add_toc: bool | DataFrame = False*, ***kwargs*)

Save the dataset to a tabular file (csv, excel, ...).

This function will trigger a computation of the dataset.

Parameters

- **ds** (*xr.Dataset or xr.DataArray*) – Dataset or DataArray to be saved. If a Dataset with more than one variable is given, the dimension "variable" must appear in one of *row*, *column* or *sheet*.
- **filename** (*str or os.PathLike*) – Name of the file to be saved.
- **output_format** (*{'csv', 'excel', ...}, optional*) – The output format. If None (default), it is inferred from the extension of *filename*. Not all possible output format are supported for inference. Valid values are any that matches a pandas.DataFrame method like "df.to_{format}".

- **row** (*str or sequence of str, optional*) – Name of the dimension(s) to use as indexes (rows). Default is all data dimensions.
- **column** (*str or sequence of str, optional*) – Name of the dimension(s) to use as columns. Default is "variable", i.e. the name of the variable(s).
- **sheet** (*str or sequence of str, optional*) – Name of the dimension(s) to use as sheet names. Only valid if the output format is excel.
- **coords** (*bool or sequence of str*) – A list of auxiliary coordinates to add to the columns (as would variables). If True, all (if any) are added.
- **col_sep** (*str,*) – Multi-columns (except in excel) and sheet names are concatenated with this separator.
- **row_sep** (*str, optional*) – Multi-index names are concatenated with this separator, except in excel. If None (default), each level is written in its own column.
- **add_toc** (*bool or DataFrame*) – A table of content to add as the first sheet. Only valid if the output format is excel. If True, `make_toc()` is used to generate the toc. The sheet name of the toc can be given through the "name" attribute of the DataFrame, otherwise "Content" is used.
- **kwargs** – Other arguments passed to the pandas function. If the output format is excel, kwargs to `pandas.ExcelWriter` can be given here as well.

xscen.io.**save_to_zarr**(*ds: Dataset*, *filename: str | PathLike*, ***, *rechunk: dict | None = None*, *zarr_kwargs: dict | None = None*, *compute: bool = True*, *encoding: dict | None = None*, *bitround: bool | int | dict = False*, *mode: str = 'f'*, *itervar: bool = False*, *timeout_cleanup: bool = True*)

Save a Dataset to Zarr format, rechunking and compressing if requested.

According to mode, removes variables that we don't want to re-compute in ds.

Parameters

- **ds** (*xr.Dataset*) – Dataset to be saved.
- **filename** (*str*) – Name of the Zarr file to be saved.
- **rechunk** (*dict, optional*) – This is a mapping from dimension name to new chunks (in any format understood by dask). Spatial dimensions can be generalized as 'X' and 'Y' which will be mapped to the actual grid type's dimension names. Rechunking is only done on *data* variables sharing dimensions with this argument.
- **zarr_kwargs** (*dict, optional*) – Additional arguments to send to_zarr()
- **compute** (*bool*) – Whether to start the computation or return a delayed object.
- **mode** (*{'f', 'o', 'a'}*) – If 'f', fails if any variable already exists. if 'o', removes the existing variables. if 'a', skip existing variables, writes the others.
- **encoding** (*dict, optional*) – If given, skipped variables are popped in place.
- **bitround** (*bool or int or dict*) – If not False, float variables are bit-rounded by dropping a certain number of bits from their mantissa, allowing for a much better compression. If an int, this is the number of bits to keep for all float variables. If a dict, a mapping from variable name to the number of bits to keep. If True, the number of bits to keep is guessed based on the variable's name, defaulting to 12, which yields a relative error of 0.012%.
- **itervar** (*bool*) – If True, (data) variables are written one at a time, appending to the zarr. If False, this function computes, no matter what was passed to kwargs.

- **timeout_cleanup** (*bool*) – If True (default) and a *xscen.scripting.TimeoutException* is raised during the writing, the variable being written is removed from the dataset as it is incomplete. This does nothing if *compute* is False.

Returns

dask.delayed object if compute=False, None otherwise.

See also:

xarray.Dataset.to_zarr

xscen.io.**subset_maxsize**(*ds: Dataset*, *maxsize_gb: float*) → list

Estimate a dataset's size and, if higher than the given limit, subset it alongside the 'time' dimension.

Parameters

- **ds** (*xr.Dataset*) – Dataset to be saved.
- **maxsize_gb** (*float*) – Target size for the NetCDF files. If the dataset is bigger than this number, it will be separated alongside the 'time' dimension.

Returns

list – List of xr.Dataset subsetted alongside 'time' to limit the filesize to the requested maximum.

xscen.io.**to_table**(*ds: Dataset | DataArray*, ***, *row: str | Sequence[str] | None = None*, *column: str | Sequence[str] | None = None*, *sheet: str | Sequence[str] | None = None*, *coords: bool | str | Sequence[str] = True*) → DataFrame | dict

Convert a dataset to a pandas DataFrame with support for multicolumns and multisheet.

This function will trigger a computation of the dataset.

Parameters

- **ds** (*xr.Dataset or xr.DataArray*) – Dataset or DataArray to be saved. If a Dataset with more than one variable is given, the dimension "variable" must appear in one of *row*, *column* or *sheet*.
- **row** (*str or sequence of str, optional*) – Name of the dimension(s) to use as indexes (rows). Default is all data dimensions.
- **column** (*str or sequence of str, optional*) – Name of the dimension(s) to use as columns. Default is "variable", i.e. the name of the variable(s).
- **sheet** (*str or sequence of str, optional*) – Name of the dimension(s) to use as sheet names.
- **coords** (*bool or str or sequence of str*) – A list of auxiliary coordinates to add to the columns (as would variables). If True, all (if any) are added.

Returns

pd.DataFrame or dict – DataFrame with a MultiIndex with levels *row* and MultiColumn with levels *column*. If *sheet* is given, the output is dictionary with keys for each unique "sheet" dimensions tuple, values are DataFrames. The DataFrames are always sorted with level priority as given in *row* and in ascending order.

xscen.reduce module

Functions to reduce an ensemble of simulations.

xscen.reduce.**build_reduction_data**(*datasets: dict | list[Dataset]*, *, *xrfreqs: list[str] | None = None*, *horizons: list[str] | None = None*) → DataArray

Construct the input required for ensemble reduction.

This will combine all variables into a single DataArray and stack all dimensions except "realization".

Parameters

- **datasets** (*Union[dict, list]*) – Dictionary of datasets in the format {"id": dataset}, or list of datasets. This can be generated by calling .to_dataset_dict() on a catalog.
- **xrfreqs** (*list of str, optional*) – List of unique frequencies across the datasets. If None, the script will attempt to guess the frequencies from the datasets' metadata or with xr.infer_freq().
- **horizons** (*list of str, optional*) – Subset of horizons on which to create the data.

Returns

xr.DataArray – 2D DataArray of dimensions "realization" and "criteria", to be used as input for ensemble reduction.

xscen.reduce.**reduce_ensemble**(*data: DataArray*, *method: str*, *kwargs: dict*)

Reduce an ensemble of simulations using clustering algorithms from xclim.ensembles.

Parameters

- **data** (*xr.DataArray*) – Selection criteria data : 2-D xr.DataArray with dimensions 'realization' and 'criteria'. These are the values used for clustering. Realizations represent the individual original ensemble members and criteria the variables/indicators used in the grouping algorithm. This data can be generated using build_reduction_data().
- **method** (*str*) – ['kkz', 'kmeans']. Clustering method.
- **kwargs** (*dict*) – Arguments to send to either xclim.ensembles.kkz_reduce_ensemble or xclim.ensembles.kmeans_reduce_ensemble

Returns

- **selected** (*xr.DataArray*) – DataArray of dimension 'realization' with the selected simulations.
- **clusters** (*dict*) – If using kmeans clustering, realizations grouped by cluster.
- **fig_data** (*dict*) – If using kmeans clustering, data necessary to call xclim.ensembles.plot_rsqprofile()

xscen.regrid module

Functions to regrid datasets.

xscen.regrid.**create_mask**(*ds: Dataset | DataArray*, *mask_args: dict*) → DataArray

Create a 0-1 mask based on incoming arguments.

Parameters

- **ds** (*xr.Dataset or xr.DataArray*) – Dataset or DataArray to be evaluated
- **mask_args** (*dict*) – Instructions to build the mask (required fields listed in the Notes).

Note:

'mask' fields:

variable: str, optional
Variable on which to base the mask, if ds_mask is not a DataArray.

where_operator: str, optional
Conditional operator such as '>'

where_threshold: str, optional
Value threshold to be used in conjunction with where_operator.

mask_nans: bool
Whether to apply a mask on NaNs.

Returns
xr.DataArray – Mask array.

xscen.regrid.**regrid_dataset**(*ds: Dataset*, *ds_grid: Dataset*, *weights_location: str | PathLike*, ***, *regridder_kwargs: dict | None = None*, *intermediate_grids: dict | None = None*, *to_level: str = 'regridded'*) → Dataset

Regrid a dataset according to weights and a reference grid.

Based on an intake_esm catalog, this function performs regridding on Zarr files.

Parameters

- **ds** (*xarray.Dataset*) – Dataset to regrid. The Dataset needs to have lat/lon coordinates. Supports a 'mask' variable compatible with ESMF standards.
- **weights_location** (*Union[str, os.PathLike]*) – Path to the folder where weight file is saved.
- **ds_grid** (*xr.Dataset*) – Destination grid. The Dataset needs to have lat/lon coordinates. Supports a 'mask' variable compatible with ESMF standards.
- **regridder_kwargs** (*dict, optional*) – Arguments to send xe.Regridder(). If it contains *skipna* or *out_chunks*, those are passed to the regridder call directly.
- **intermediate_grids** (*dict, optional*) – This argument is used to do a regridding in many steps, regridding to regular grids before regridding to the final ds_grid. This is useful when there is a large jump in resolution between ds and ds grid. The format is a nested dictionary shown in Notes. If None, no intermediary grid is used, there is only a regrid from ds to ds_grid.
- **to_level** (*str*) – The processing level to assign to the output. Defaults to 'regridded'

Returns
xarray.Dataset – Regridded dataset

Notes

intermediate_grids =

**{'name_of_inter_grid_1': {'cf_grid_2d': {arguments for util.cf_grid_2d
},'regridder_kwargs':{arguments for xe.Regridder}},**
'name_of_inter_grid_2': dictionary_as_above}

See also:

`xesmf.regridder`, `xesmf.util.cf_grid_2d`

xscen.scripting module

A collection of various convenience objects and functions to use in scripts.

exception xscen.scripting.**TimeoutException**(*seconds: int*, *task: str = ''*, ***kwargs*)

Bases: `Exception`

An exception raised with a timeout occurs.

class xscen.scripting.**measure_time**(*name: str | None = None*, *cpu: bool = False*, *logger: ~logging.Logger = <Logger xscen.scripting (INFO)>*)

Bases: `object`

Context for timing a code block.

Parameters

- **name** (*str, optional*) – A name to give to the block being timed, for meaningful logging.
- **cpu** (*boolean*) – If True, the CPU time is also measured and logged.
- **logger** (*logging.Logger, optional*) – The logger object to use when sending Info messages with the measured time. Defaults to a logger from this module.

xscen.scripting.**move_and_delete**(*moving: list[list[str | PathLike]]*, *pcat:* ProjectCatalog, *deleting: list[str | PathLike] | None = None*, *copy: bool = False*)

First, move files, then update the catalog with new locations. Finally, delete directories.

This function can be used at the end of for loop in a workflow to clean temporary files.

Parameters

- **moving** (*list of lists of str or os.PathLike*) – list of lists of path of files to move, following the format: [[source 1, destination1], [source 2, destination2],...]
- **pcat** (*ProjectCatalog*) – Catalog to update with new destinations
- **deleting** (*list of str or os.PathLike, optional*) – list of directories to be deleted including all contents and recreated empty. E.g. the working directory of a workflow.
- **copy** (*bool, optional*) – If True, copy directories instead of moving them.

xscen.scripting.**save_and_update**(*ds: Dataset*, *pcat:* ProjectCatalog, *path: str | PathLike | None = None*, *file_format: str | None = None*, *build_path_kwargs: dict | None = None*, *save_kwargs: dict | None = None*, *update_kwargs: dict | None = None*)

Construct the path, save and delete.

This function can be used after each task of a workflow.

Parameters

- **ds** (*xr.Dataset*) – Dataset to save.
- **pcat** (*ProjectCatalog*) – Catalog to update after saving the dataset.
- **path** (*str or os.pathlike, optional*) – Path where to save the dataset. If the string contains variables in curly bracket. They will be filled by catalog attributes. If None, the *catutils.build_path* fonction will be used to create a path.
- **file_format** (*{'nc', 'zarr'}*) – Format of the file. If None, look for the following in order: build_path_kwargs['format'], a suffix in path, ds.attrs['cat:format']. If nothing is found, it will default to zarr.
- **build_path_kwargs** (*dict, optional*) – Arguments to pass to *build_path*.
- **save_kwargs** (*dict, optional*) – Arguments to pass to *save_to_netcdf* or *save_to_zarr*.
- **update_kwargs** (*dict, optional*) – Arguments to pass to *update_from_ds*.

xscen.scripting.**send_mail**(*, *subject: str*, *msg: str*, *to: str | None = None*, *server: str = '127.0.0.1'*, *port: int = 25*, *attachments: list[tuple[str, Figure | PathLike] | Figure | PathLike] | None = None*) → None

Send email.

Email a single address through a login-less SMTP server. The default values of server and port should work out-of-the-box on Ouranos's systems.

Parameters

- **subject** (*str*) – Subject line.
- **msg** (*str*) – Main content of the email. Can be UTF-8 and multi-line.
- **to** (*str, optional*) – Email address to which send the email. If None (default), the email is sent to "{os.getlogin()}@{os.uname().nodename}". On unix systems simply put your real email address in *$HOME/.forward* to receive the emails sent to this local address.
- **server** (*str*) – SMTP server url. Defaults to 127.0.0.1, the local host. This function does not try to log-in.
- **port** (*int*) – Port of the SMTP service on the server. Defaults to 25, which is usually the default port on unix-like systems.
- **attachments** (*list of paths or matplotlib figures or tuples of a string and a path or figure, optional*) – List of files to attach to the email. Elements of the list can be paths, the mimetypes of those is guessed and the files are read and sent. Elements can also be matplotlib Figures which are send as png image (savefig) with names like "Figure00.png". Finally, elements can be tuples of a filename to use in the email and the attachment, handled as above.

Returns

None

xscen.scripting.**send_mail_on_exit**(*, *subject: str | None = None*, *msg_ok: str | None = None*, *msg_err: str | None = None*, *on_error_only: bool = False*, *skip_ctrlc: bool = True*, ***mail_kwargs*) → None

Send an email with content depending on how the system exited.

This function is best used by registering it with *atexit*. Calls `send_mail()`.

Parameters

- **subject** (*str, optional*) – Email subject. Will be appended by "Success", "No errors" or "Failure" depending on how the system exits.

- **msg_ok** (*str, optional*) – Content of the email if the system exists successfully.
- **msg_err** (*str, optional*) – Content of the email id the system exists with a non-zero code or with an error. The message will be appended by the exit code or with the error traceback.
- **on_error_only** (*boolean*) – Whether to only send an email on a non-zero/error exit.
- **skip_ctrlc** (*boolean*) – If True (default), exiting with a KeyboardInterrupt will not send an email.
- **mail_kwargs** – Other arguments passed to *send_mail()*. The *to* argument is necessary for this function to work.

Returns
None

Example

Send an eamil titled "Woups" upon non-successful program exit. We assume the *to* field was given in the config.

```
>>> import atexit
>>> atexit.register(send_mail_on_exit, subject="Woups", on_error_only=True)
```

xscen.scripting.**skippable**(*seconds: int = 2*, *task: str = ''*, *logger: Logger | None = None*)

Skippable context manager.

When CTRL-C (SIGINT, KeyboardInterrupt) is sent within the context, this catches it, prints to the log and gives a timeout during which a subsequent interruption will stop the script. Otherwise, the context exits normally.

This is meant to be used within a loop so that we can skip some iterations:

```
for i in iterable:
    with skippable(2, i):
        some_skippable_code()
```

Parameters

- **seconds** (*int*) – Number of seconds to wait for a second CTRL-C.
- **task** (*str*) – A name for the skippable task, to have an explicit script.
- **logger** (*logging.Logger, optional*) – The logger to use when printing the messages. The interruption signal is notified with ERROR, while the skipping is notified with INFO. If not given (default), a brutal print is used.

xscen.scripting.**timeout**(*seconds: int*, *task: str = ''*)

Timeout context manager.

Only one can be used at a time, this is not multithread-safe : it cannot be used in another thread than the main one, but multithreading can be used in parallel.

Parameters

- **seconds** (*int*) – Number of seconds after which the context exits with a TimeoutException. If None or negative, no timeout is set and this context does nothing.
- **task** (*str, optional*) – A name to give to the task, allowing a more meaningful exception.

xscen.spatial module

Spatial tools.

xscen.spatial.**creep_fill**(*da: DataArray*, *w: DataArray*) → DataArray

Creep fill using pre-computed weights.

Parameters

- **da** (*DataArray*) – A DataArray sharing the dimensions with the one used to compute the weights. It can have other dimensions. Dask is supported as long as there are no chunks over the creeped dims.
- **w** (*DataArray*) – The result of *creep_weights*.

Returns

xarray.DataArray, same shape as *da*, but values filled according to *w*.

Examples

```
>>> w = creep_weights(da.isel(time=0).notnull(), n=1)
>>> da_filled = creep_fill(da, w)
```

xscen.spatial.**creep_weights**(*mask: DataArray*, *n: int = 1*, *mode: str = 'clip'*) → DataArray

Compute weights for the creep fill.

The output is a sparse matrix with the same dimensions as *mask*, twice.

Parameters

- **mask** (*DataArray*) – A boolean DataArray. False values are candidates to the filling. Usually they represent missing values (*mask = da.notnull()*). All dimensions are creep filled.
- **n** (*int*) – The order of neighbouring to use. 1 means only the adjacent grid cells are used.
- **mode** (*{'clip', 'wrap'}*) – If a cell is on the edge of the domain, *mode='wrap'* will wrap around to find neighbours.

Returns

DataArray – Weights. The dot product must be taken over the last N dimensions.

xscen.spatial.**subset**(*ds: Dataset*, *region: dict | None = None*, ***, *name: str | None = None*, *method: str | None = None*, *tile_buffer: float = 0*, ***kwargs*) → Dataset

Subset the data to a region.

Either creates a slice and uses the .sel() method, or customizes a call to clisops.subset() that allows for an automatic buffer around the region.

Parameters

- **ds** (*xr.Dataset*) – Dataset to be subsetted.
- **region** (*dict*) – Deprecated argument that is there for legacy reasons and will be abandoned eventually.
- **name** (*str, optional*) – Used to rename the 'cat:domain' attribute.
- **method** (*str*) – ['gridpoint', 'bbox', shape','sel'] If the method is *sel*, this is not a call to clisops but only a subsetting with the xarray .sel() fonction.

- **tile_buffer** (*float*) – For ['bbox', shape'], uses an approximation of the grid cell size to add a buffer around the requested region. This differs from clisops' 'buffer' argument in subset_shape().
- **kwargs** (*dict*) – Arguments to be sent to clisops. If the method is *sel*, the keys are the dimensions to subset and the values are turned into a slice.

Returns

xr.Dataset – Subsetted Dataset.

See also:

clisops.core.subset.subset_gridpoint, clisops.core.subset.subset_bbox, clisops.core.subset.subset_shape

xscen.testing module

Testing utilities for xscen.

xscen.testing.**datablock_3d**(*values: ndarray*, *variable: str*, *x: str*, *x_start: float*, *y: str*, *y_start: float*, *x_step: float = 0.1*, *y_step: float = 0.1*, *start: str = '7/1/2000'*, *freq: str = 'D'*, *units: str | None = None*, *as_dataset: bool = False*) → DataArray | Dataset

Create a generic timeseries object based on pre-defined dictionaries of existing variables.

Parameters

- **values** (*np.ndarray*) – The values to be assigned to the variable. Dimensions are interpreted [T, Y, X].
- **variable** (*str*) – The variable name.
- **x** (*str*) – The name of the x coordinate.
- **x_start** (*float*) – The starting value of the x coordinate.
- **y** (*str*) – The name of the y coordinate.
- **y_start** (*float*) – The starting value of the y coordinate.
- **x_step** (*float*) – The step between x values.
- **y_step** (*float*) – The step between y values.
- **start** (*str*) – The starting date of the time coordinate.
- **freq** (*str*) – The frequency of the time coordinate.
- **units** (*str, optional*) – The units of the variable. If None, the units are inferred from the variable name.
- **as_dataset** (*bool*) – If True, return a Dataset, else a DataArray.

xscen.testing.**fake_data**(*nyears: int*, *nx: int*, *ny: int*, *rand_type: str = 'random'*, *seed: int = 0*, *amplitude: float = 1.0*, *offset: float = 0.0*) → ndarray

Generate fake data for testing.

Parameters

- **nyears** (*int*) – Number of years (365 days) to generate.
- **nx** (*int*) – Number of x points.
- **ny** (*int*) – Number of y points.

- **rand_type** (*str*) – Type of random data to generate. Options are: - "random": random data with no structure. - "tas": temperature-like data with a yearly half-sine cycle.
- **seed** (*int*) – Random seed.
- **amplitude** (*float*) – Amplitude of the random data.
- **offset** (*float*) – Offset of the random data.

Returns
np.ndarray – Fake data.

xscen.utils module

Common utilities to be used in many places.

xscen.utils.**add_attr**(*ds: Dataset | DataArray*, *attr: str*, *new: str*, ***fmt*)

Add a formatted translatable attribute to a dataset.

xscen.utils.**change_units**(*ds: Dataset*, *variables_and_units: dict*) → Dataset

Change units of Datasets to non-CF units.

Parameters

- **ds** (*xr.Dataset*) – Dataset to use
- **variables_and_units** (*dict*) – Description of the variables and units to output

Returns
xr.Dataset

See also:

xclim.core.units.convert_units_to, xclim.core.units.rate2amount

xscen.utils.**clean_up**(*ds: Dataset*, ***, *variables_and_units: dict | None = None*, *convert_calendar_kwargs: dict | None = None*, *missing_by_var: dict | None = None*, *maybe_unstack_dict: dict | None = None*, *round_var: dict | None = None*, *common_attrs_only: dict | list[Dataset | str | PathLike] | None = None*, *common_attrs_open_kwargs: dict | None = None*, *attrs_to_remove: dict | None = None*, *remove_all_attrs_except: dict | None = None*, *add_attrs: dict | None = None*, *change_attr_prefix: str | None = None*, *to_level: str | None = None*) → Dataset

Clean up of the dataset.

It can:

- convert to the right units using xscen.finalize.change_units
- convert the calendar and interpolate over missing dates
- call the xscen.common.maybe_unstack function
- remove a list of attributes
- remove everything but a list of attributes
- add attributes
- change the prefix of the catalog attrs

in that order.

Parameters

- **ds** (*xr.Dataset*) – Input dataset to clean up
- **variables_and_units** (*dict, optional*) – Dictionary of variable to convert. eg. {'tasmax': 'degC', 'pr': 'mm d-1'}
- **convert_calendar_kwargs** (*dict, optional*) – Dictionary of arguments to feed to xclim.core.calendar.convert_calendar. This will be the same for all variables. If missing_by_vars is given, it will override the 'missing' argument given here. Eg. {target': default, 'align_on': 'random'}
- **missing_by_var** (*dict, optional*) – Dictionary where the keys are the variables and the values are the argument to feed the *missing* parameters of the xclim.core.calendar.convert_calendar for the given variable with the *convert_calendar_kwargs*. When the value of an entry is 'interpolate', the missing values will be filled with NaNs, then linearly interpolated over time.
- **maybe_unstack_dict** (*dict, optional*) – Dictionary to pass to xscen.common.maybe_unstack function. The format should be: {'coords': path_to_coord_file, 'rechunk': {'time': -1 }, 'stack_drop_nans': True}.
- **round_var** (*dict, optional*) – Dictionary where the keys are the variables of the dataset and the values are the number of decimal places to round to
- **common_attrs_only** (*dict, list of datasets, or list of paths, optional*) – Dictionnary of datasets or list of datasets, or path to NetCDF or Zarr files. Keeps only the global attributes that are the same for all datasets and generates a new id.
- **common_attrs_open_kwargs** (*dict, optional*) – Dictionary of arguments for xarray.open_dataset(). Used with common_attrs_only if given paths.
- **attrs_to_remove** (*dict, optional*) – Dictionary where the keys are the variables and the values are a list of the attrs that should be removed. For global attrs, use the key 'global'. The element of the list can be exact matches for the attributes name or use the same substring matching rules as intake_esm: - ending with a '*' means checks if the substring is contained in the string - starting with a '^' means check if the string starts with the substring. eg. {'global': ['unnecessary note', 'cell*'], 'tasmax': 'old_name'}
- **remove_all_attrs_except** (*dict, optional*) – Dictionary where the keys are the variables and the values are a list of the attrs that should NOT be removed, all other attributes will be deleted. If None (default), nothing will be deleted. For global attrs, use the key 'global'. The element of the list can be exact matches for the attributes name or use the same substring matching rules as intake_esm: - ending with a '*' means checks if the substring is contained in the string - starting with a '^' means check if the string starts with the substring. eg. {'global': ['necessary note', '^cat:'], 'tasmax': 'new_name'}
- **add_attrs** (*dict, optional*) – Dictionary where the keys are the variables and the values are a another dictionary of attributes. For global attrs, use the key 'global'. eg. {'global': {'title': 'amazing new dataset'}, 'tasmax': {'note': 'important info about tasmax'}}
- **change_attr_prefix** (*str, optional*) – Replace "cat:" in the catalog global attrs by this new string
- **to_level** (*str, optional*) – The processing level to assign to the output.

Returns

xr.Dataset – Cleaned up dataset

See also:

`xclim.core.calendar.convert_calendar`

xscen.utils.**date_parser**(*date: str | datetime | Timestamp | datetime | Period*, ***, *end_of_period: bool | str = False*, *out_dtype: str = 'datetime'*, *strtime_format: str = '%Y-%m-%d'*, *freq: str = 'H'*) → str | Period | Timestamp

Return a datetime from a string.

Parameters

- **date** (*str, cftime.datetime, pd.Timestamp, datetime.datetime, pd.Period*) – Date to be converted
- **end_of_period** (*bool or str*) – If 'Y' or 'M', the returned date will be the end of the year or month that contains the received date. If True, the period is inferred from the date's precision, but *date* must be a string, otherwise nothing is done.
- **out_dtype** (*str*) – Choices are 'datetime', 'period' or 'str'
- **strtime_format** (*str*) – If out_dtype=='str', this sets the strftime format
- **freq** (*str*) – If out_dtype=='period', this sets the frequency of the period.

Returns

pd.Timestamp, pd.Period, str – Parsed date

xscen.utils.**get_cat_attrs**(*ds: Dataset | DataArray | dict*, *prefix: str = 'cat:'*, *var_as_str=False*) → dict

Return the catalog-specific attributes from a dataset or dictionary.

Parameters

- **ds** (*xr.Dataset, dict*) – Dataset to be parsed. If a dictionary, it is assumed to be the attributes of the dataset (ds.attrs).
- **prefix** (*str*) – Prefix automatically generated by intake-esm. With xscen, this should be 'cat:'
- **var_as_str** (*bool*) – If True, 'variable' will be returned as a string if there is only one.

Returns

dict – Compilation of all attributes in a dictionary.

xscen.utils.**maybe_unstack**(*ds: Dataset*, *coords: str | None = None*, *rechunk: dict | None = None*, *stack_drop_nans: bool = False*) → Dataset

If stack_drop_nans is True, unstack and rechunk.

Parameters

- **ds** (*xr.Dataset*) – Dataset to unstack.
- **coords** (*str, optional*) – Path to a dataset containing the coords to unstack (and only those).
- **rechunk** (*dict, optional*) – If not None, rechunk the dataset after unstacking.
- **stack_drop_nans** (*bool*) – If True, unstack the dataset and rechunk it. If False, do nothing.

Returns

xr.Dataset – Unstacked dataset.

xscen.utils.**minimum_calendar**(**calendars*) → str

Return the minimum calendar from a list.

Uses the hierarchy: 360_day < noleap < standard < all_leap, and returns one of those names.

xscen.utils.**natural_sort**(*_list: list[str]*)

For strings of numbers. alternative to sorted() that detects a more natural order.

e.g. [r3i1p1, r1i1p1, r10i1p1] is sorted as [r1i1p1, r3i1p1, r10i1p1] instead of [r10i1p1, r1i1p1, r3i1p1]

xscen.utils.**publish_release_notes**(*style: str = 'md'*, *file: PathLike | StringIO | TextIO | None = None*, *changes: str | PathLike = None*) → str | None

Format release history in Markdown or ReStructuredText.

Parameters

- **style** (*{"rst", "md"}*) – Use ReStructuredText (*rst*) or Markdown (*md*) formatting. Default: Markdown.
- **file** (*{os.PathLike, StringIO, TextIO, None}*) – If provided, prints to the given file-like object. Otherwise, returns a string.
- **changes** (*{str, os.PathLike}, optional*) – If provided, manually points to the file where the changelog can be found. Assumes a relative path otherwise.

Returns

str, optional

Notes

This function exists solely for development purposes. Adapted from xclim.testing.utils.publish_release_notes.

xscen.utils.**stack_drop_nans**(*ds: Dataset*, *mask: DataArray*, ***, *new_dim: str = 'loc'*, *to_file: str | None = None*) → Dataset

Stack dimensions into a single axis and drops indexes where the mask is false.

Parameters

- **ds** (*xr.Dataset*) – A dataset with the same coords as *mask*.
- **mask** (*xr.DataArray*) – A boolean DataArray with True on the points to keep. Mask will be loaded within this function.
- **new_dim** (*str*) – The name of the new stacked dim.
- **to_file** (*str, optional*) – A netCDF filename where to write the stacked coords for use in *unstack_fill_nan*. If given a string with {shape} and {domain}, the formatting will fill them with the original shape of the dataset and the global attributes 'cat:domain'. If None (default), nothing is written to disk. It is recommended to fill this argument in the config. It will be parsed automatically. E.g.:

  utils:

  stack_drop_nans:
  to_file: /some_path/coords/coords_{domain}_{shape}.nc

  unstack_fill_nan:
  coords: /some_path/coords/coords_{domain}_{shape}.nc

Returns

xr.Dataset – Same as *ds*, but all dimensions of mask have been stacked to a single *new_dim*. Indexes where mask is False have been dropped.

See also:

unstack_fill_nan
The inverse operation.

xscen.utils.**standardize_periods**(*periods: list[str] | list[list[str]] | None*, *multiple: bool = True*) → list[str] | list[list[str]] | None

Reformats the input to a list of strings, ['start', 'end'], or a list of such lists.

Parameters

- **periods** (*list of str or list of lists of str, optional*) – The period(s) to standardize. If None, return None.
- **multiple** (*bool*) – If True, return a list of periods, otherwise return a single period.

xscen.utils.**translate_time_chunk**(*chunks: dict*, *calendar: str*, *timesize*) → dict

Translate chunk specification for time into a number.

-1 translates to *timesize* 'Nyear' translates to N times the number of days in a year of calendar *calendar*.

xscen.utils.**unstack_dates**(*ds: Dataset*, *seasons: dict[int, str] | None = None*, *new_dim: str = 'season'*, *winter_starts_year: bool = False*)

Unstack a multi-season timeseries into a yearly axis and a season one.

Parameters

- **ds** (*xr.Dataset or DataArray*) – The xarray object with a "time" coordinate. Only supports monthly or coarser frequencies. The time axis must be complete and regular (*xr.infer_freq(ds.time)* doesn't fail).
- **seasons** (*dict, optional*) – A dictionary from month number (as int) to a season name. If not given, it is guessed from the time coord's frequency. See notes.
- **new_dim** (*str*) – The name of the new dimension.
- **winter_starts_year** (*bool*) – If True, the year of winter (DJF) is built from the year of January, not December. i.e. DJF made from [Dec 1980, Jan 1981, and Feb 1981] will be associated with the year 1981, not 1980.

Returns

xr.Dataset or DataArray – Same as ds but the time axis is now yearly (AS-JAN) and the seasons are along the new dimension.

Notes

When *season* is None, the inferred frequency determines the new coordinate:

- For MS, the coordinates are the month abbreviations in english (JAN, FEB, etc.)
- For ?QS-? and other ?MS frequencies, the coordinates are the initials of the months in each season. Ex: QS-DEC (with winter_starts_year=True) : DJF, MAM, JJA, SON.
- For YS or AS-JAN, the new coordinate has a single value of "annual".
- For ?AS-? frequencies, the new coordinate has a single value of "annual-{anchor}", were "anchor" is the abbreviation of the first month of the year. Ex: AS-JUL -> "annual-JUL".

xscen.utils.**unstack_fill_nan**(*ds: Dataset*, ***, *dim: str = 'loc'*, *coords: str | PathLike | Sequence[str | PathLike] | dict | None = None*)

Unstack a Dataset that was stacked by `stack_drop_nans()`.

Parameters

- **ds** (*xr.Dataset*) – A dataset with some dims stacked by *stack_drop_nans*.
- **dim** (*str*) – The dimension to unstack, same as *new_dim* in *stack_drop_nans*.

- **coords** (*Sequence of strings, Mapping of str to array, str, optional*) – If a sequence : if the dataset has coords along *dim* that are not original dimensions, those original dimensions must be listed here. If a dict : a mapping from the name to the array of the coords to unstack If a str : a filename to a dataset containing only those coords (as coords). If given a string with {shape} and {domain}, the formatting will fill them with the original shape of the dataset (that should have been store in the attributes of the stacked dimensions) by *stack_drop_nans* and the global attributes 'cat:domain'. It is recommended to fill this argument in the config. It will be parsed automatically. E.g.:

  utils:

  stack_drop_nans:
  to_file: /some_path/coords/coords_{domain}_{shape}.nc

  unstack_fill_nan:
  coords: /some_path/coords/coords_{domain}_{shape}.nc

  If None (default), all coords that have *dim* a single dimension are used as the new dimensions/coords in the unstacked output. Coordinates will be loaded within this function.

Returns
xr.Dataset – Same as *ds*, but *dim* has been unstacked to coordinates in *coords*. Missing elements are filled according to the defaults of *fill_value* of `xarray.Dataset.unstack()`.

xscen.utils.**update_attr**(*ds: Dataset | DataArray*, *attr: str*, *new: str*, *others: Sequence[Dataset | DataArray] | None = None*, ***fmt*) → Dataset | DataArray

Format an attribute referencing itself in a translatable way.

Parameters

- **ds** (*Dataset or DataArray*) – The input object with the attribute to update.
- **attr** (*str*) – Attribute name.
- **new** (*str*) – New attribute as a template string. It may refer to the old version of the attribute with the "{attr}" field.
- **others** (*Sequence of Datasets or DataArrays*) – Other objects from which we can extract the attribute *attr*. These can be referenced as "{attrXX}" in *new*, where XX is the based-1 index of the other source in *others*. If they don't have the *attr* attribute, an empty string is sent to the string formatting. See notes.
- **fmt** – Other formatting data.

Returns
ds, but updated with the new version of *attr*, in each of the activated languages.

Notes

This is meant for constructing attributes by extending a previous version or combining it from different sources. For example, given a *ds* that has *long_name="Variability"*:

```
>>> update_attr(ds, "long_name", _("Mean of {attr}"))
```

Will update the "long_name" of *ds* with *long_name="Mean of Variability"*. The use of *_(...)* allows the detection of this string by the translation manager. The function will be able to add a translatable version of the string for each activated language, for example adding a *long_name_fr="Moyenne de Variabilité"* (assuming a *long_name_fr* was present on the initial *ds*).

If the new attribute is an aggregation from multiple sources, these can be passed in *others*.

```
>>> update_attr(
...     ds0,
...     "long_name",
...     _("Addition of {attr} and {attr1}, divided by {attr2}"),
...     others=[ds1, ds2],
... )
```

Here, *ds0* will have it's *long_name* updated with the passed string, where *attr1* is the *long_name* of *ds1* and *attr2* the *long_name* of *ds2*. The process will be repeated for each localized *long_name* available on *ds0*. For example, if *ds0* has a *long_name_fr*, the template string is translated and filled with the *long_name_fr* attributes of *ds0*, *ds1* and *ds2*. If the latter don't exist, the english version is used instead.

PYTHON MODULE INDEX

X

INDEX

N

P

R

S

T

U

W

X