DejaVu Serif DejaVu Sans DejaVu Sans Mono

xclim Documentation

Release 0.37.0

Travis Logan

Jun 18, 2022

CONTENTS:

`xclim` is a library of functions to compute climate indices from observations or model simulations. It is built using xarray and can benefit from the parallelization handling provided by dask. Its objective is to make it as simple as possible for users to compute indices from large climate datasets and for scientists to write new indices with very little boilerplate.

For applications where meta-data and missing values are important to get right, xclim provides a class for each index that validates inputs, checks for missing values, converts units and assigns metadata attributes to the output. This also provides a mechanism for users to customize the indices to their own specifications and preferences.

xclim currently provides over 50 indices related to mean, minimum and maximum daily temperature, daily precipitation, streamflow and sea ice concentration.

CHAPTER

ONE

INSTALLATION

1.1 Stable release

To install xclim via pip, run this command in your terminal:

```
$ pip install xclim
```

This is the preferred method to install xclim, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.2 Anaconda release

For ease of installation across operating systems, we also offer an Anaconda Python package hosted on conda-forge. This version tends to be updated at around the same frequency as the pip library, but can lag by a few days at times.

To install the xclim Anaconda binary, run this command in your terminal:

```
$ conda install -c conda-forge xclim
```

1.3 Extra dependencies

To improve performance of xclim, we highly recommend you also install flox (see: flox API). This package integrates into xarray and significantly improves the performance of the grouping and resampling algorithms, especially when using *dask* on large datasets.

We also recommend using the subsetting tools in clisops (see: clisops.core.subset API) for spatial manipulation of geospatial data.

xclim is regularly tested against the main development branches of a handful of key base libraries (xarray, cftime, flox, pint). For convenience, these libraries can be installed alongside *xclim* using the following pip-installable recipe:

```
$ pip install -e ".[upstream]"
```

1.4 From sources

Warning: For Python3.10+ users: Many of the required scientific libraries do not currently have wheels that support the latest python. In order to ensure that installation of xclim doesn't fail, we suggest installing the *Cython* module before installing xclim in order to compile necessary libraries from source packages.

The sources for xclim can be downloaded from the Github repo.

You can either clone the public repository:

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

Or download the tarball:

```
$ curl -OL https://github.com/Ouranosinc/xclim/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

Alternatively, you can also install a local development copy via pip:

```
$ pip install -e .[dev]
```

1.5 Creating a Conda environment

To create a conda development environment including all xclim dependencies, enter the following command from within your cloned repo:

```
$ conda create -n my_xclim_env python=3.8 --file=environment.yml
$ conda activate my_xclim_env
(my_xclim_env) $ pip install .[dev]
```

CHAPTER

TWO

BASIC USAGE

2.1 Climate indicator computations

`xclim` is a library of climate indicators that operate on xarray `DataArray` objects.

`xclim` provides two layers of computations, one responsible for computations and units handling (the computation layer, the **indices**), and the other responsible for input health checks and metadata formatting (the CF layer, refering to the Climate and Forecast convention, the **indicators**). Functions from the computation layer are found in `xclim.indices`, while indicator objects from the CF layer are found in *realm* modules (`xclim.atmos`, `xclim.land` and `xclim.seaIce`). Users should always use the indicators, and maybe revert to indices as a last resort if the indicator machinery becomes too heavy for their special edge case.

To use xclim in a project, import both `xclim` and `xarray`.

[1]:
```
from __future__ import annotations

import xarray as xr

import xclim
from xclim.testing import open_dataset
```

Indice calculations are performed by opening a netCDF-like file, accessing the variable of interest, and calling the indice function, which returns a new DataArray.

For this example, we'll first open a demonstration dataset storing surface air temperature and compute the number of growing degree days (the sum of degrees above a certain threshold) at the monthly frequency.

[2]:
```
# ds = xr.open_dataset("your_file.nc")
ds = open_dataset("ERA5/daily_surface_cancities_1990-1993.nc")
ds.tas
```

[2]:
```
<xarray.DataArray 'tas' (location: 5, time: 1461)>
array([[277.49966, 270.44736, 273.5631 , ..., 259.30075, 267.44043, 264.0009 ],
       [272.3179 , 268.01813, 273.50452, ..., 249.57759, 258.23706, 260.20535],
       [245.21338, 252.72534, 248.18385, ..., 235.18086, 236.17192, 243.2071 ],
       [270.79147, 263.67996, 257.4426 , ..., 257.80548, 269.45105, 261.2271 ],
       [279.71753, 278.1774 , 279.41824, ..., 280.08725, 280.65396, 280.92868]],
      dtype=float32)
Coordinates:
    lat       (location) float32 ...
  * location  (location) object 'Halifax' 'Montréal' ... 'Saskatoon' 'Victoria'
    lon       (location) float32 ...
  * time      (time) datetime64[ns] 1990-01-01 1990-01-02 ... 1993-12-31
```

(continues on next page)

(continued from previous page)

```
Attributes:
    standard_name:  air_temperature
    long_name:      Mean daily surface temperature
    units:          K
    cell_methods:   time: mean within days
```

[3]:
```
gdd = xclim.atmos.growing_degree_days(tas=ds.tas, thresh="10.0 degC", freq="YS")
gdd
```

[3]:
```
<xarray.DataArray 'growing_degree_days' (location: 5, time: 4)>
array([[7.9897247e+02, 7.2488672e+02, 6.4941925e+02, 6.7033386e+02],
       [1.2330164e+03, 1.3716892e+03, 1.1340271e+03, 1.2288167e+03],
       [8.0615845e+00, 2.7421051e+01, 1.0251160e+00, 2.0045013e+01],
       [9.3481873e+02, 1.0134860e+03, 7.2482220e+02, 6.3551764e+02],
       [6.2461761e+02, 5.3345679e+02, 6.3453369e+02, 5.9410144e+02]],
      dtype=float32)
Coordinates:
  * time      (time) datetime64[ns] 1990-01-01 1991-01-01 1992-01-01 1993-01-01
    lat       (location) float32 44.5 45.5 63.75 52.0 48.5
  * location  (location) object 'Halifax' 'Montréal' ... 'Saskatoon' 'Victoria'
    lon       (location) float32 -63.5 -73.5 -68.5 -106.8 -123.2
Attributes:
    units:          K days
    cell_methods:   time: mean within days time: sum over days
    history:        [2022-06-18 02:36:08] growing_degree_days: GROWING_DEGREE...
    standard_name:  integral_of_air_temperature_excess_wrt_time
    long_name:      Growing degree days above 10.0 degc
    description:    Annual growing degree days above 10.0 degc.
```

This computation was made using the `growing_degree_days` **indicator**. The same computation could be made through the **indice**. You can see how the metadata is alot poorer here.

[4]:
```
gdd = xclim.indices.growing_degree_days(tas=ds.tas, thresh="10.0 degC", freq="YS")
gdd
```

[4]:
```
<xarray.DataArray 'tas' (location: 5, time: 4)>
array([[7.9897247e+02, 7.2488672e+02, 6.4941925e+02, 6.7033386e+02],
       [1.2330164e+03, 1.3716892e+03, 1.1340271e+03, 1.2288167e+03],
       [8.0615845e+00, 2.7421051e+01, 1.0251160e+00, 2.0045013e+01],
       [9.3481873e+02, 1.0134860e+03, 7.2482220e+02, 6.3551764e+02],
       [6.2461761e+02, 5.3345679e+02, 6.3453369e+02, 5.9410144e+02]],
      dtype=float32)
Coordinates:
  * time      (time) datetime64[ns] 1990-01-01 1991-01-01 1992-01-01 1993-01-01
    lat       (location) float32 44.5 45.5 63.75 52.0 48.5
  * location  (location) object 'Halifax' 'Montréal' ... 'Saskatoon' 'Victoria'
    lon       (location) float32 -63.5 -73.5 -68.5 -106.8 -123.2
Attributes:
    units:    K d
```

The call to `xclim.indices.growing_degree_days` first checked that the input variable units were units of temperature, ran the computation, then set the output's units to the appropriate unit (here `K d` or kelvin days). As you can see, the **indicator** returned the same output, but with more metadata, it also performed more checks as explained below.

`growing_degree_days` makes most sense with **daily input**, but could theoritically accept other source frequencies. The computational layer (*indice*) assumes that users have checked that the input data has the expected temporal frequency and has no missing values. However, no checks are performed, so the output data could be wrong. That's why it's always safer to use **``Indicator``** objects from the CF layer, as done in the following section.

New unit handling paradigm in xclim 0.24 for indices

As of xclim 0.24, the paradigm in unit handling has changed slightly. Now, indices are written in order to be more flexible as to the sampling frequency and units of the data. You *can* use `growing_degree_days` on, for example, the 6-hourly data. The ouput will then be in degree-hour units (`K h`). Moreover, all units, even when untouched by the calculation, will be reformatted to a CF-compliant symbol format. This was made to ensure consistency between all indices.

Very few indices will convert their output to a specific units, rather it is the dimensionality that will be consistent. The *Unit handling* page goes in more details on how unit conversion can easily be done.

This doesn't apply to **Indicators**. Those will always output data in a specific unit, the one listed in the `Indicators.cf_attrs` metadata dictionnary.

Finally, as almost all indices, the function takes a `freq` argument to specify over what time period it is computed. These are called "Offset Aliases" and are the same as the resampling string arguments. Valid arguments are detailed in panda's doc (note that aliases involving "business" notions are not supported by `xarray` and thus could raises issues in xclim.

2.2 Health checks and metadata attributes

Indicator instances from the CF layer are found in modules bearing the name of the computational realm in which its input variables are found: `xclim.atmos`, `xclim.land` and `xclim.seaIce`. These objects from the CF layer run sanity checks on the input variables and set output's metadata according to CF-convention when they apply. Some of the checks involve:

- Identifying periods where missing data significantly impacts the calculation and omits calculations for those periods. Those are called "missing methods" and are detailed in section *Health checks*.
- Appending process history and maintaining the historical provenance of file metadata.
- Writing Climate and Forecast Convention compliant metadata based on the variables and indices calculated.

Those modules are best used for producing NetCDF that will be shared with users. See *Climate Indicators* for a list of available indicators.

If we run the `growing_degree_days` indicator over a non daily dataset, we'll be warned that the input data is not daily. That is, running `xclim.atmos.growing_degree_days(ds.air, thresh='10.0 degC', freq='MS')` will fail with a `ValidationError`:

[5]:
```
ds6h = xr.tutorial.open_dataset("air_temperature")
xr.infer_freq(ds6h.time)  # Show that it is not daily
```

[5]:
```
'6H'
```

[6]:
```
gdd = xclim.atmos.growing_degree_days(tas=ds6h.tas, thresh="10.0 degC", freq="MS")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [6], in <cell line: 1>()
----> 1 gdd = xclim.atmos.growing_degree_days(tas=ds6h.tas, thresh="10.0 degC", freq="MS
↪")

File ~/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xarray/core/common.py:239, in AttrAccessMixin.__getattr__(self, name)
    237         with suppress(KeyError):
    238             return source[name]
--> 239 raise AttributeError(
    240     f"{type(self).__name__!r} object has no attribute {name!r}"
    241 )

AttributeError: 'Dataset' object has no attribute 'tas'
```

Resampling to a daily frequency and running the same indicator succeeds, but we still get warnings from the CF metadata checks.

```
[7]: daily_ds = ds6h.resample(time="D").mean(keep_attrs=True)
     gdd = xclim.atmos.growing_degree_days(daily_ds.air, thresh="10.0 degC", freq="YS")
```

```
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/core/cfchecks.py:41: UserWarning: Variable does not have a `cell_
↪methods` attribute.
  _check_cell_methods(
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/core/cfchecks.py:45: UserWarning: Variable does not have a `standard_
↪name` attribute.
  check_valid(vardata, "standard_name", data["standard_name"])
```

To suppress the CF validation warnings in the following, we will set xclim to send them to the log, instead of raising a warning or an error. We also could have set `data_validation='warn'` to be able to run the indicator on non-daily data. These options are set globally or within a context with *set_options*.

The missing method which determines if a period should be considered missing or not can be controlled through the `check_missing` option, globally or contextually. The main missing methods also have options that can be modified.

```
[8]: with xclim.set_options(
         check_missing="pct",
         missing_options={"pct": dict(tolerance=0.1)},
         cf_compliance="log",
     ):
         # Change the missing method to "percent", instead of the default "any"
         # Set the tolerance to 10%, periods with more than 10% of missing data
         #     in the input will be masked in the ouput.
         gdd = xclim.atmos.growing_degree_days(daily_ds.air, thresh="10.0 degC", freq="MS")
```

Some indicators also expose time-selection arguments as `**indexer` keywords. This allows to run the indice on a subset of the time coordinates, for example only on a specific season, month, or between two dates in every year. It relies on the *select_time* function. Some indicators will simply select the time period and run the calculations, while others will smartly perform the selection at the right time, when the order of operation makes a difference. All will pass the `indexer` kwargs to the missing value handling ensuring that the missing values *outside* the valid time period are **not** considered.

The next example computes the annual sum of growing degree days over 10 °C, but only considering days from the 1st of April to the 30th of September.

```
[9]: with xclim.set_options(cf_compliance="log"):
         gdd = xclim.atmos.growing_degree_days(
             tas=daily_ds.air, thresh="10 degC", freq="YS", date_bounds=("04-01", "09-30")
         )
     gdd
```

```
[9]: <xarray.DataArray 'growing_degree_days' (time: 2, lat: 25, lon: 53)>
     array([[[0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ...,
              0.0000000e+00, 0.0000000e+00, 0.0000000e+00],
             [0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ...,
              0.0000000e+00, 0.0000000e+00, 0.0000000e+00],
             [3.3140015e+01, 5.0820099e+01, 6.6547607e+01, ...,
              0.0000000e+00, 0.0000000e+00, 0.0000000e+00],
             ...,
             [2.7736938e+03, 2.6248127e+03, 2.5183259e+03, ...,
              2.6201809e+03, 2.5202236e+03, 2.4362007e+03],
             [2.8073425e+03, 2.7539409e+03, 2.6544858e+03, ...,
              2.6141130e+03, 2.6077131e+03, 2.5585962e+03],
             [2.8185554e+03, 2.8164487e+03, 2.7658499e+03, ...,
              2.6862107e+03, 2.6818704e+03, 2.6931643e+03]],

            [[0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ...,
              0.0000000e+00, 0.0000000e+00, 0.0000000e+00],
             [0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ...,
              0.0000000e+00, 0.0000000e+00, 0.0000000e+00],
             [1.0225220e+00, 5.5400085e+00, 1.0475037e+01, ...,
              0.0000000e+00, 0.0000000e+00, 0.0000000e+00],
             ...,
             [2.8183235e+03, 2.6905312e+03, 2.6107827e+03, ...,
              2.5506511e+03, 2.4474639e+03, 2.3652024e+03],
             [2.8695332e+03, 2.8242588e+03, 2.7269099e+03, ...,
              2.5259944e+03, 2.5199478e+03, 2.4677590e+03],
             [2.8881079e+03, 2.8856880e+03, 2.8283704e+03, ...,
              2.5869858e+03, 2.5948555e+03, 2.6111182e+03]]], dtype=float32)
     Coordinates:
       * time     (time) datetime64[ns] 2013-01-01 2014-01-01
       * lat      (lat) float32 75.0 72.5 70.0 67.5 65.0 ... 25.0 22.5 20.0 17.5 15.0
       * lon      (lon) float32 200.0 202.5 205.0 207.5 ... 322.5 325.0 327.5 330.0
     Attributes:
         units:           K days
         cell_methods:     time: sum over days
         history:         [2022-06-18 02:36:10] growing_degree_days: GROWING_DEGREE...
         standard_name:   integral_of_air_temperature_excess_wrt_time
         long_name:       Growing degree days above 10 degc
         description:     Annual growing degree days above 10 degc.
```

Finally, xclim also allows to call indicators using datasets and variable names.

```
[10]: with xclim.set_options(cf_compliance="log"):
          gdd = xclim.atmos.growing_degree_days(
              tas="air", thresh="10.0 degC", freq="MS", ds=daily_ds
```

(continues on next page)

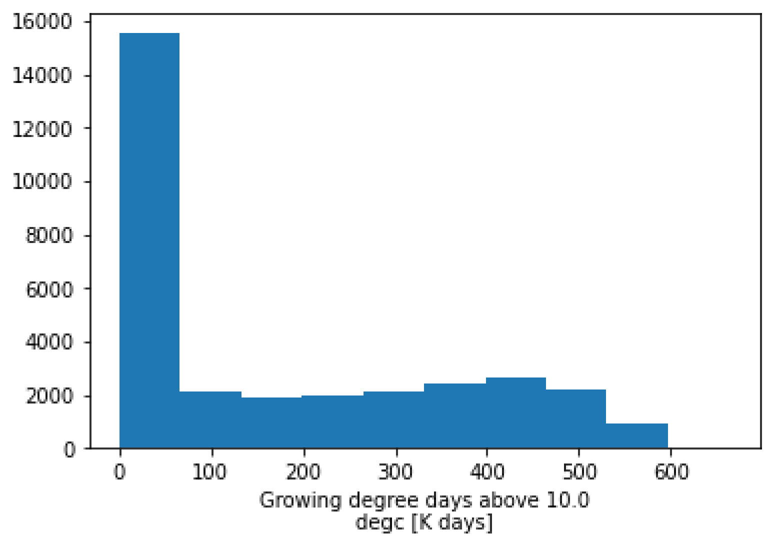
(continued from previous page)

```
    )

    # variable names default to xclim names, so we can even do this:
    renamed_daily_ds = daily_ds.rename(air="tas")
    gdd = xclim.atmos.growing_degree_days(
        thresh="10.0 degC", freq="MS", ds=renamed_daily_ds
    )
```

2.3 Graphics

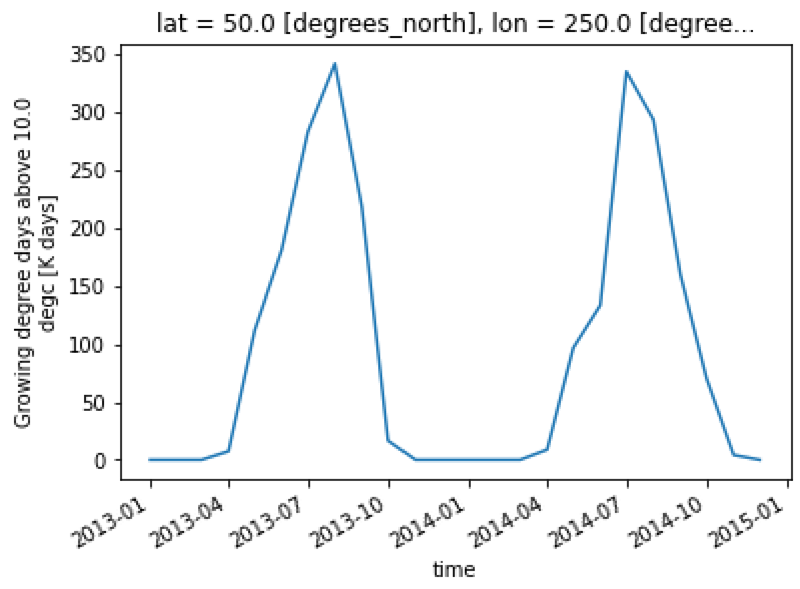
[11]:
```
import matplotlib.pyplot

%matplotlib inline

# Summary statistics histogram
gdd.plot()
```

[11]:
```
(array([15532.,  2079.,  1861.,  1935.,  2137.,  2416.,  2675.,  2202.,
          931.,    32.]),
 array([  0.      ,  66.32573, 132.65146, 198.9772 , 265.30292, 331.62866,
        397.9544 , 464.28012, 530.60583, 596.9316 , 663.2573 ],
       dtype=float32),
 <BarContainer object of 10 artists>)
```



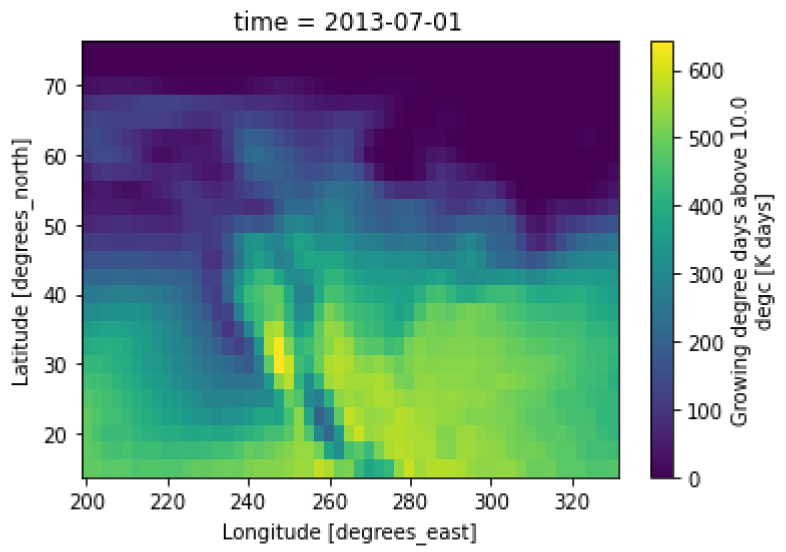
[12]:
```
# Show time series at a given geographical coordinate
gdd.isel(lon=20, lat=10).plot()
```

[12]:
```
[<matplotlib.lines.Line2D at 0x7fdcbd7d7610>]
```



[13]:
```
# Show spatial pattern at a specific time period
gdd.sel(time="2013-07").plot()
```

[13]:
```
<matplotlib.collections.QuadMesh at 0x7fdcbd328bb0>
```



For more examples, see the directions suggested by xarray's plotting documentation

To save the data as a new NetCDF, use `to_netcdf`.

[14]:
```
gdd.to_netcdf("monthly_growing_degree_days_data.nc")
```

It's possible to save Dataset objects to other file formats. For more information see: xarray's documentation

CHAPTER THREE

EXAMPLES

3.1 Workflow Examples

xclim is built on very powerful multiprocessing and distributed computation libraries, notably xarray and dask.

xarray is a python package making it easy to work with n-dimensional arrays. It labels axes with their names [time, lat, lon, level] instead of indices [0,1,2,3], reducing the likelihood of bugs and making the code easier to understand. One of the key strengths of xarray is that it knows how to deal with non-standard calendars (we're looking at you, "360_days") and can easily resample daily time series to weekly, monthly, seasonal or annual periods. Finally, xarray is tightly inegrated with dask, a package that can automatically parallelize operations.

The following are a few examples to consult when using xclim to subset netCDF arrays and compute climate indicators, taking advantage of the parallel processing capabilities offered by xarray and dask. For more information about these projects, please see their documentation pages:

- xarray documentation
- dask documentation

3.1.1 Environment configuration

[ ]:
```
# Imports for xclim and xarray
from __future__ import annotations

import numpy as np
import xarray as xr

import xclim as xc

xr.set_options(display_style="html")

import tempfile

# File handling libraries
import time
from pathlib import Path

# Output folder
output_folder = Path(tempfile.mkdtemp())
```

3.1.2 Setting up the Dask client: parallel processing

In this example, we are using the **dask.distributed** submodule. This is not installed by default in a basic `xclim` installation. Be sure to add `distributed` to your Python installation before setting up parallel processing operations!

First we create a pool of workers that will wait for jobs. The `xarray` library will automatically connect to these workers and and dispatch them jobs that can be run in parallel.

The dashboard link lets you see in real time how busy those workers are.

- dask distributed documentation

This step is not mandatory as dask will fall back to its "single machine scheduler" if a `Client` is not created. However, this default scheduler doesn't allow you to set the number of threads or a memory limit and doesn't start the dashboard, which can be quite useful to understand your task's progress.

[ ]:
```
from distributed import Client

# Depending on your workstation specifications, you may need to adjust these values.
# On a single machine, n_workers=1 is usually better.
client = Client(n_workers=1, threads_per_worker=4, memory_limit="4GB")
client
```

```
<Client: 'tcp://127.0.0.1:44837' processes=1 threads=4, memory=4.00 GB>
```

3.1.3 Creating xarray datasets

To open a netCDF file with `xarray`, we use `xr.open_dataset(<path to file>)`. By default, the entire file is stored in one chunk, so there is no parallelism. To trigger parallel computations, we need to explicitly specify the **chunk size**.

In this example, instead of opening a local file, we pass an *OPeNDAP* url to xarray. It retrieves the data automatically. Notice also that opening the dataset is quite fast. In fact, the data itself has not been downloaded yet, only the coordinates and the metadata. The downloads will be triggered only when the values need to be accessed directly.

`dask`'s parallelism is based on memory chunks. We need to tell `xarray` to split our netCDF array into chunks of a given size, and operations on each chunk of the array will automatically be dispatched to the workers.

[ ]:
```
data_url = "https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/dodsC/datasets/
↪simulations/bias_adjusted/cmip5/ouranos/cb-oura-1.0/day_ACCESS1-3_historical+rcp85_
↪r1i1p1_na10kgrid_qm-moving-50bins-detrend_1950-2100.ncml"
```

[ ]:
```
# Chunking in memory along the time dimension.
# Note that the data type is a 'dask.array'. xarray will automatically use client␣
↪workers.
ds = xr.open_dataset(
    data_url,
    chunks={"time": 365, "lat": 168, "lon": 150},
    drop_variables=["ts", "time_vectors"],
```

(continues on next page)

(continued from previous page)

```
)
print(ds)
```

```
<xarray.Dataset>
Dimensions:  (lat: 320, lon: 797, time: 55152)
Coordinates:
  * lat      (lat) float32 66.62331 66.53998 66.45665 ... 40.12437 40.04104
  * lon      (lon) float32 -120.79394 -120.71061 ... -54.54659 -54.46326
  * time     (time) datetime64[ns] 1950-01-01 1950-01-02 ... 2100-12-31
Data variables:
    tasmin   (time, lat, lon) float32 dask.array<chunksize=(365, 168, 150), meta=np.
↪ndarray>
    tasmax   (time, lat, lon) float32 dask.array<chunksize=(365, 168, 150), meta=np.
↪ndarray>
    pr       (time, lat, lon) float32 dask.array<chunksize=(365, 168, 150), meta=np.
↪ndarray>
Attributes:
    Conventions:                 CF-1.5
    title:                       Ouranos standard ensemble of bias-adjusted cl...
    history:                     CMIP5 compliant file produced from raw ACCESS...
    institution:                 Ouranos Consortium on Regional Climatology an...
    source:                      ACCESS1-3 2011. Atmosphere: AGCM v1.0 (N96 gr...
    driving_model:               ACCESS1-3
    driving_experiment:          historical,rcp85
    institute_id:                Ouranos
    type:                        GCM
    processing:                  bias_adjusted
    dataset_description:         https://www.ouranos.ca/publication-scientifiq...
    bias_adjustment_method:      1D-Quantile Mapping
    bias_adjustment_reference:   http://doi.org/10.1002/2015JD023890
    project_id:                  CMIP5
    licence_type:                permissive
    terms_of_use:                Terms of use at https://www.ouranos.ca/climat...
    attribution:                 Use of this dataset should be acknowledged as...
    frequency:                   day
    modeling_realm:              atmos
    target_dataset:              CANADA : ANUSPLIN interpolated Canada daily 3...
    target_dataset_references:   CANADA : https://doi.org/10.1175/2011BAMS3132...
    driving_institution:         Commonwealth Scientific and Industrial Resear...
    driving_institute_id:        CSIRO-BOM
```

```
[ ]: print(ds.tasmin.chunks)
```

```
((365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365,␣
↪365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365,␣
↪365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365,␣
↪365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365,␣
↪365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365,␣
↪365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365,␣
↪365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365,␣
↪365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365,␣
↪365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 37), (168,␣
↪152), (150, 150, 150, 150, 150, 47))
```

3.1.4 Multi-file datasets

NetCDF files are often split into periods to keep file size manageable. A single dataset can be split in dozens of individual files. `xarray` has a function `open_mfdataset` that can open and aggregate a list of files and construct a unique *logical* dataset. `open_mfdataset` can aggregate files over coordinates (time, lat, lon) and variables.

- Note that opening a multi-file dataset automatically chunks the array (one chunk per file).
- Note also that because `xarray` reads every file metadata to place it in a logical order, it can take a while to load.

[ ]:
```
## Create multi-file data & chunks
# ds = xr.open_mfdataset('/path/to/files*.nc')
```

3.1.5 Subsetting and selecting data with xarray

Usually, xclim users are encouraged to use the subsetting utilities of the clisops package. Here, we will reduce the size of our data using the methods implemented in xarray (docs here).

[ ]:
```
ds2 = ds.sel(lat=slice(50, 45), lon=slice(-70, -65), time=slice("2090", "2100"))
print(ds2.tasmin)
```

```
<xarray.DataArray 'tasmin' (time: 4017, lat: 60, lon: 60)>
dask.array<getitem, shape=(4017, 60, 60), dtype=float32, chunksize=(365, 60, 60),␣
↪chunktype=numpy.ndarray>
Coordinates:
  * lat      (lat) float32 49.95731 49.87398 49.79065 ... 45.12417 45.04084
  * lon      (lon) float32 -69.96264 -69.87931 -69.79598 ... -65.1295 -65.04617
  * time     (time) datetime64[ns] 2090-01-01 2090-01-02 ... 2100-12-31
Attributes:
    long_name:      air_temperature
    standard_name:  air_temperature
    units:          K
    _ChunkSizes:    [256  16  16]
```

[ ]:
```
ds3 = ds.sel(lat=46.8, lon=-71.22, method="nearest").sel(time="1993")
print(ds3.tasmin)
```

3.1.6 Climate index calculation & resampling frequencies

`xclim` has two layers for the calculation of indicators. The bottom layer is composed of a list of functions that take one or more `xarray.DataArray`'s as input and return an `xarray.DataArray` as output. You'll find these functions in `xclim.indices`. The indicator's logic is contained in this function, as well as some unit handling, but it doesn't perform any data consistency checks (like if the time frequency is daily), and doesn't not adjust the metadata of the output array.

The second layer are class instances that you'll find organized by *realm*. So far, there are three realms available in `xclim.atmos`, `xclim.seaIce` and `xclim.land`, the first one being the most exhaustive. Before running computations, these classes check if the input data is a daily average of the expected variable:

1. If an indicator expects a daily mean and you pass it a daily max, a `warning` will be raised.

2. After the computation, it also checks the number of values per period to make sure there are not missing values or `NaN` in the input data. If there are, the output is going to be set to `NaN`. Ex. : If the indicator performs a yearly resampling but there are only 350 non-`NaN` values in one given year in the input data, that year's output will be `NaN`.
3. The output units are set correctly as well as other properties of the output array, complying as much as possible with CF conventions.

For new users, we suggest you use the classes found in `xclim.atmos` and others. If you know what you're doing and you want to circumvent the built-in checks, then you can use the `xclim.indices` directly.

Almost all `xclim` indicators convert daily data to lower time frequencies, such as seasonal or annual values. This is done using `xarray.DataArray.resample` method. Resampling creates a grouped object over which you apply a reduction operation (e.g. mean, min, max). The list of available frequency is given in the link below, but the most often used are:

- `YS`: annual starting in January
- `YS-JUL`: annual starting in July
- `MS`: monthly
- `QS-DEC`: seasonal starting in December

More info about this specification can be found in pandas' documentation

Note - not all offsets in the link are supported by cftime objects in `xarray`.

In the example below, we're computing the **annual maximum temperature of the daily maximum temperature (tx_max)**.

```
[ ]: out = xc.atmos.tx_max(ds2.tasmax, freq="YS")
     print(out)
```

```
/home/phobos/Python/xclim/xclim/indicators/atmos/_temperature.py:87: UserWarning:␣
↪Variable does not have a `cell_methods` attribute.
  cfchecks.check_valid(tasmax, "cell_methods", "*time: maximum within days*")
```

```
<xarray.DataArray 'tx_max' (time: 11, lat: 60, lon: 60)>
dask.array<where, shape=(11, 60, 60), dtype=float32, chunksize=(1, 60, 60),␣
↪chunktype=numpy.ndarray>
Coordinates:
  * time     (time) datetime64[ns] 2090-01-01 2091-01-01 ... 2100-01-01
  * lat      (lat) float32 49.95731 49.87398 49.79065 ... 45.12417 45.04084
  * lon      (lon) float32 -69.96264 -69.87931 -69.79598 ... -65.1295 -65.04617
Attributes:
    long_name:      Maximum daily maximum temperature
    standard_name:  air_temperature
    units:          K
    _ChunkSizes:    [256  16  16]
    cell_methods:    time: maximum within days time: maximum over days
    xclim_history:  [2021-02-15 17:08:48] tx_max: tx_max(tasmax=<array>, freq...
    description:    Annual maximum of daily maximum temperature.
```

If you execute the cell above, you'll see that this operation is quite fast. This a feature coming from dask. Read *Lazy computation* further down.

Comparison of atmos vs indices modules

Using the `xclim.indices` module performs not checks and only fills the `units` attribute.

```
[ ]: out = xc.indices.tx_days_above(ds2.tasmax, thresh="30 C", freq="YS")
     print(out)
```

```
<xarray.DataArray 'tasmax' (time: 11, lat: 60, lon: 60)>
dask.array<mul, shape=(11, 60, 60), dtype=int64, chunksize=(1, 60, 60), chunktype=numpy.
↪ndarray>
Coordinates:
  * time     (time) datetime64[ns] 2090-01-01 2091-01-01 ... 2100-01-01
  * lat      (lat) float32 49.95731 49.87398 49.79065 ... 45.12417 45.04084
  * lon      (lon) float32 -69.96264 -69.87931 -69.79598 ... -65.1295 -65.04617
Attributes:
    units:    d
```

With `xclim.atmos`, checks are performed and many CF-compliant attributes are added:

```
[ ]: out = xc.atmos.tx_days_above(ds2.tasmax, thresh="30 C", freq="YS")
     print(out)
```

```
<xarray.DataArray 'tx_days_above' (time: 11, lat: 60, lon: 60)>
dask.array<where, shape=(11, 60, 60), dtype=float64, chunksize=(1, 60, 60),␣
↪chunktype=numpy.ndarray>
Coordinates:
  * time     (time) datetime64[ns] 2090-01-01 2091-01-01 ... 2100-01-01
  * lat      (lat) float32 49.95731 49.87398 49.79065 ... 45.12417 45.04084
  * lon      (lon) float32 -69.96264 -69.87931 -69.79598 ... -65.1295 -65.04617
Attributes:
    units:          days
    cell_methods:    time: maximum within days time: sum over days
    xclim_history:  [2021-02-15 17:08:49] tx_days_above: tx_days_above(tasmax...
    standard_name:  number_of_days_with_air_temperature_above_threshold
    long_name:      Number of days with tmax > 30 c
    description:    Annual number of days where daily maximum temperature exc...
```

```
/home/phobos/Python/xclim/xclim/indicators/atmos/_temperature.py:87: UserWarning:␣
↪Variable does not have a `cell_methods` attribute.
  cfchecks.check_valid(tasmax, "cell_methods", "*time: maximum within days*")
```

```
[ ]: # We have created an xarray data-array - We can insert this into an output xr.Dataset␣
     ↪object with a copy of the original dataset global attrs
     dsOut = xr.Dataset(attrs=ds2.attrs)

     # Add our climate index as a data variable to the dataset
     dsOut[out.name] = out
     print(dsOut)
```

```
<xarray.Dataset>
Dimensions:         (lat: 60, lon: 60, time: 11)
Coordinates:
  * time            (time) datetime64[ns] 2090-01-01 2091-01-01 ... 2100-01-01
  * lat             (lat) float32 49.95731 49.87398 ... 45.12417 45.04084
  * lon             (lon) float32 -69.96264 -69.87931 ... -65.1295 -65.04617
```

(continues on next page)

(continued from previous page)

```
Data variables:
    tx_days_above  (time, lat, lon) float64 dask.array<chunksize=(1, 60, 60), meta=np.
↪ndarray>
Attributes:
    Conventions:                  CF-1.5
    title:                        Ouranos standard ensemble of bias-adjusted cl...
    history:                      CMIP5 compliant file produced from raw ACCESS...
    institution:                  Ouranos Consortium on Regional Climatology an...
    source:                       ACCESS1-3 2011. Atmosphere: AGCM v1.0 (N96 gr...
    driving_model:                ACCESS1-3
    driving_experiment:           historical,rcp85
    institute_id:                 Ouranos
    type:                         GCM
    processing:                   bias_adjusted
    dataset_description:          https://www.ouranos.ca/publication-scientifiq...
    bias_adjustment_method:       1D-Quantile Mapping
    bias_adjustment_reference:    http://doi.org/10.1002/2015JD023890
    project_id:                   CMIP5
    licence_type:                 permissive
    terms_of_use:                 Terms of use at https://www.ouranos.ca/climat...
    attribution:                  Use of this dataset should be acknowledged as...
    frequency:                    day
    modeling_realm:               atmos
    target_dataset:               CANADA : ANUSPLIN interpolated Canada daily 3...
    target_dataset_references:    CANADA : https://doi.org/10.1175/2011BAMS3132...
    driving_institution:          Commonwealth Scientific and Industrial Resear...
    driving_institute_id:         CSIRO-BOM
```

3.1.7 *Lazy* computation - Nothing has been computed so far !

If you look at the output of those operations, they're identified as `dask.array` objects. What happens is that `dask` creates a chain of operations that when executed, will yield the values we want. We have thus far only created a schedule of tasks with a small preview and not done any actual computations. You can trigger computations by using the `load` or `compute` method, or writing the output to disk via `to_netcdf`. Of course, calling `.plot()` will also trigger the computation.

[ ]:
```
%%time
output_file = output_folder / "test_tx_max.nc"
dsOut.to_netcdf(output_file)
```

```
CPU times: user 1.1 s, sys: 74.4 ms, total: 1.17 s
Wall time: 14.4 s
```

(Times may of course vary depending on the machine and the Client settings)

Performance tips

Optimizing the chunk size

You can improve performance by being smart about chunk sizes. If chunks are too small, there is a lot of time lost in overhead. If chunks are too large, you may end up exceeding the individual worker memory limit.

```
[ ]: print(ds2.chunks["time"])
```

```
(330, 365, 365, 365, 365, 365, 365, 365, 365, 365, 365, 37)
```

```
[ ]: # rechunk data in memory for the entire grid
ds2c = ds2.chunk(chunks={"time": 4 * 365})
print(ds2c.chunks["time"])
```

```
(1460, 1460, 1097)
```

```
[ ]: %%time
out = xc.atmos.tx_max(ds2c.tasmax, freq="YS")
dsOut = xr.Dataset(data_vars=None, coords=out.coords, attrs=ds.attrs)
dsOut[out.name] = out

output_file = output_folder / "test_tx_max.nc"
dsOut.to_netcdf(output_file)
```

```
/home/phobos/Python/xclim/xclim/indicators/atmos/_temperature.py:87: UserWarning:␣
↪Variable does not have a `cell_methods` attribute.
  cfchecks.check_valid(tasmax, "cell_methods", "*time: maximum within days*")
```

```
CPU times: user 582 ms, sys: 75.1 ms, total: 657 ms
Wall time: 5.42 s
```

Loading the data in memory

If the dataset is relatively small, it might be more efficient to simply load the data into the memory and use numpy arrays instead of dask arrays.

```
[ ]: ds4 = ds3.load()
```

3.1.8 Unit handling in `xclim`

A lot of effort has been placed into automatic handling of input data units. `xclim` will automatically detect the input variable(s) units (e.g. °C versus °K or mm/s versus mm/day etc.) and adjust on-the-fly in order to calculate indices in the consistent manner. This comes with the obvious caveat that input data requires metadata attribute for units.

In the example below, we compute weekly total precipitation in mm using inputs of mm/s and mm/d. As you see, the output is identical.

```
[ ]: # Compute with the original mm s-1 data
out1 = xc.atmos.precip_accumulation(ds4.pr, freq="MS")
```

(continues on next page)

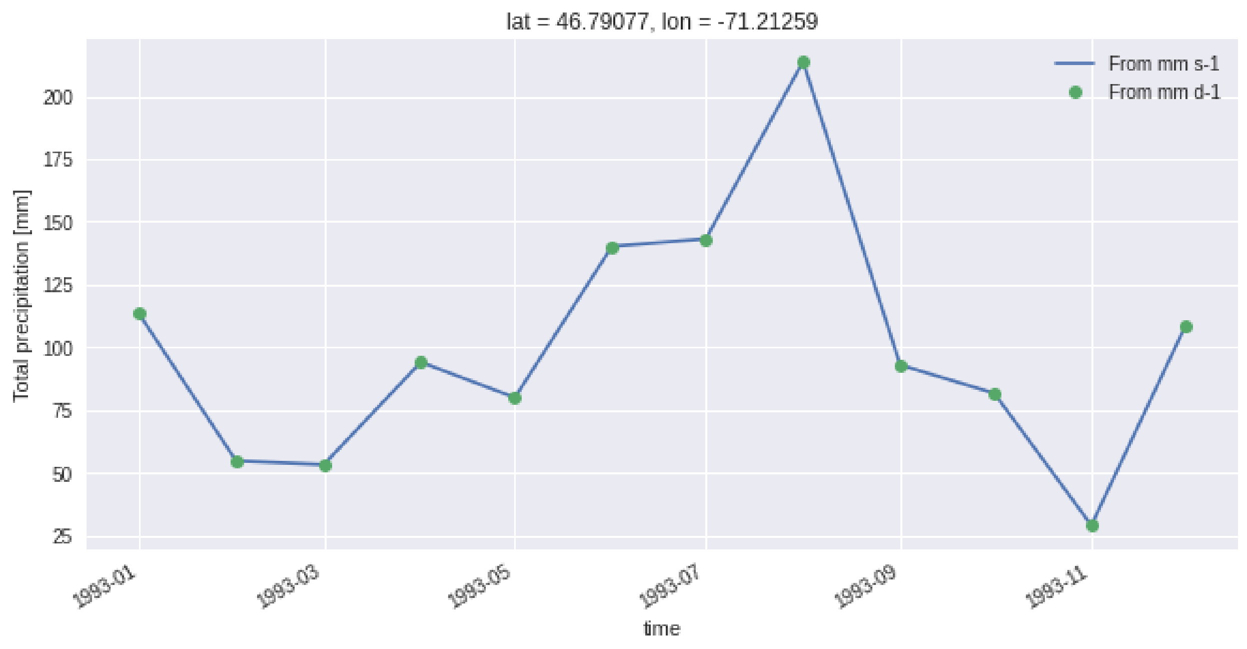
(continued from previous page)

```
# Create a copy of the data converted to mm d-1
pr_mmd = ds4.pr * 3600 * 24
pr_mmd.attrs["units"] = "mm d-1"
out2 = xc.atmos.precip_accumulation(pr_mmd, freq="MS")
```

[ ]:
```
# import plotting stuff
import matplotlib.pyplot as plt

%matplotlib inline
plt.style.use("seaborn")
plt.rcParams["figure.figsize"] = (11, 5)
```

[ ]:
```
plt.figure()
out1.plot(label="From mm s-1", linestyle="-")
out2.plot(label="From mm d-1", linestyle="none", marker="o")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fb6e8360b50>
```



Threshold indices

xclim unit handling also applies to threshold indicators. Users can provide threshold in units of choice and xclim will adjust automatically. For example determining the number of days with tasmax > 20°C users can define a threshold input of '20 C' or '20 degC' even if input data is in Kelvin. Alernatively users can even provide a threshold in Kelvin '293.15 K' (if they really wanted to).

[ ]:
```
# Create a copy of the data converted to C
tasmax_C = ds4.tasmax - 273.15
tasmax_C.attrs["units"] = "C"
```

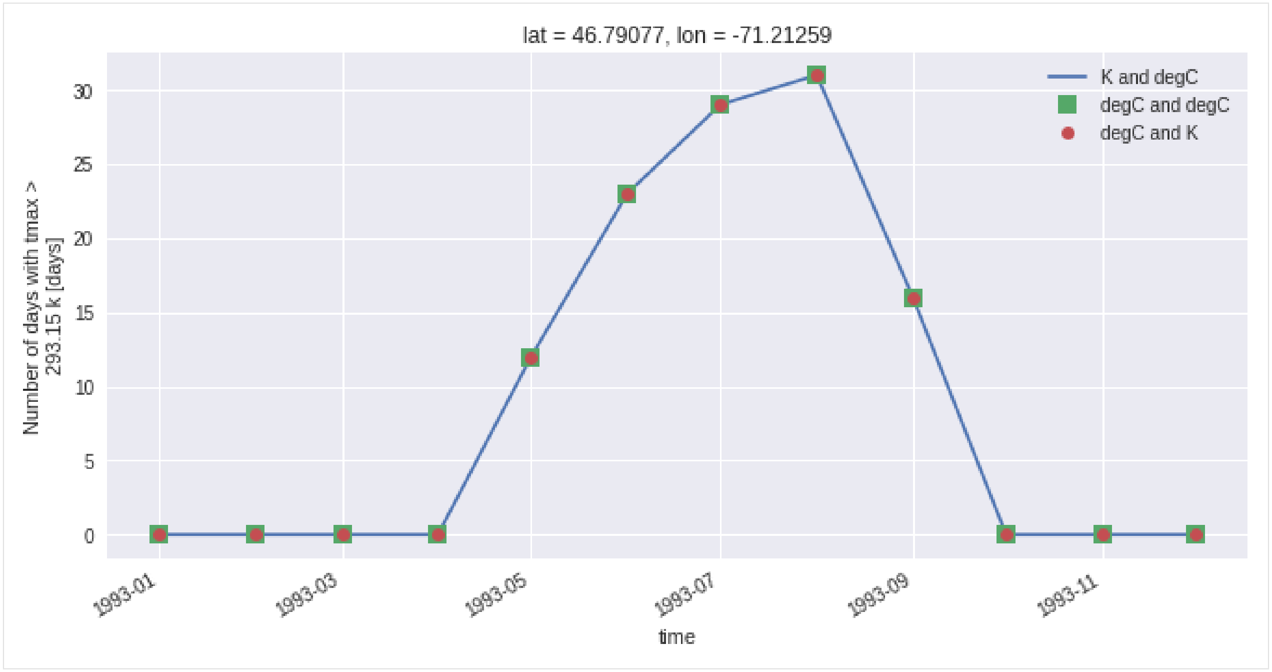
(continues on next page)

(continued from previous page)

```
# Using Kelvin data, threshold in Celsius
out1 = xc.atmos.tx_days_above(ds4.tasmax, thresh="20 C", freq="MS")

# Using Celsius data
out2 = xc.atmos.tx_days_above(tasmax_C, thresh="20 C", freq="MS")

# Using Celsius but with threshold in Kelvin
out3 = xc.atmos.tx_days_above(tasmax_C, thresh="293.15 K", freq="MS")

# Plot and see that it's all identical:
plt.figure()
out1.plot(label="K and degC", linestyle="-")
out2.plot(label="degC and degC", marker="s", markersize=10, linestyle="none")
out3.plot(label="degC and K", marker="o", linestyle="none")
plt.legend()
```

```
/home/phobos/Python/xclim/xclim/indicators/atmos/_temperature.py:87: UserWarning:␣
↪Variable does not have a `cell_methods` attribute.
  cfchecks.check_valid(tasmax, "cell_methods", "*time: maximum within days*")
/home/phobos/Python/xclim/xclim/indicators/atmos/_temperature.py:87: UserWarning:␣
↪Variable does not have a `cell_methods` attribute.
  cfchecks.check_valid(tasmax, "cell_methods", "*time: maximum within days*")
/home/phobos/Python/xclim/xclim/indicators/atmos/_temperature.py:88: UserWarning:␣
↪Variable does not have a `standard_name` attribute.
  cfchecks.check_valid(tasmax, "standard_name", "air_temperature")
/home/phobos/Python/xclim/xclim/indicators/atmos/_temperature.py:87: UserWarning:␣
↪Variable does not have a `cell_methods` attribute.
  cfchecks.check_valid(tasmax, "cell_methods", "*time: maximum within days*")
/home/phobos/Python/xclim/xclim/indicators/atmos/_temperature.py:88: UserWarning:␣
↪Variable does not have a `standard_name` attribute.
  cfchecks.check_valid(tasmax, "standard_name", "air_temperature")
```

```
<matplotlib.legend.Legend at 0x7fb6e8190340>
```



3.2 Ensembles

An important aspect of climate models is that they are run multiple times with some initial perturbations to see how they replicate the natural variability of the climate. Through *xclim.ensembles*, xclim provides an easy interface to compute ensemble statistics on different members. Most methods perform checks and conversion on top of simpler `xarray` methods, providing an easier interface to use.

3.2.1 create_ensemble

Our first step is to create an ensemble. This methods takes a list of files defining the same variables over the same coordinates and concatenates them into one dataset with an added dimension `realization`.

Using `xarray` a very simple way of creating an ensemble dataset would be :

```
import xarray
xarray.open_mfdataset(files, concat_dim='realization')
```

However, this is only successful when the dimensions of all the files are identical AND only if the calendar type of each netcdf file is the same

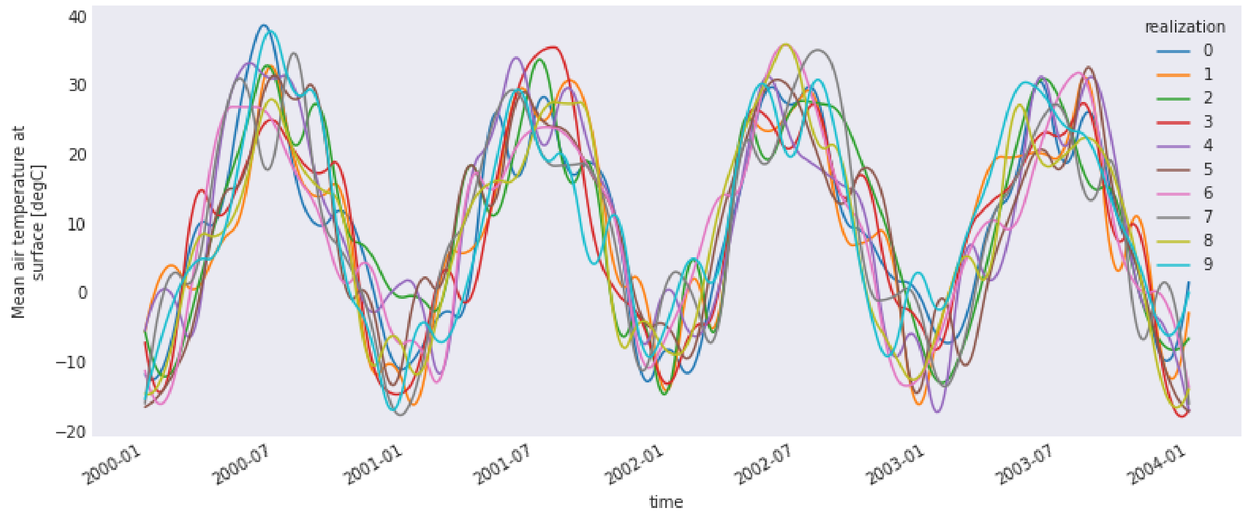
xclim's `create_ensemble()` method overcomes these constraints selecting the common time period to all files and assigns a standard calendar type to the dataset.

Input netcdf files still require equal spatial dimension size (e.g. lon, lat dimensions). If input data contains multiple cftime calendar types they must not be at daily frequency.

Given files all named `ens_tas_m[member number].nc`, we use `glob` to get a list of all those files.

[2]:
```
import glob

import xarray as xr

import xclim as xc

# Set display to HTML sytle (for fancy output)
xr.set_options(display_style="html", display_width=50)

import matplotlib.pyplot as plt

%matplotlib inline

from xclim import ensembles

ens = ensembles.create_ensemble(glob.glob("ens_tas_m*.nc")).load()
ens.close()
```

[3]:
```
plt.style.use("seaborn-dark")
plt.rcParams["figure.figsize"] = (13, 5)
ens.tas.plot(hue="realization")
plt.show()
```



[4]:
```
ens.tas  # Attributes of the first dataset to be opened are copied to the final output
```

[4]:
```
<xarray.DataArray 'tas' (realization: 10,
                         time: 1461)>
array([[-11.78754982, -11.93558326, -12.06795501, ...,   0.11432034,
          0.81866502,   1.54719821],
       [ -5.65574993,  -5.1495412 ,  -4.6568133 , ...,  -4.41886858,
         -3.66212057,  -2.87112245],
       [ -5.52218344,  -5.98167853,  -6.42443583, ...,  -6.88336783,
         -6.73370639,  -6.57630236],
       ...,
       [-15.99596777, -15.13037749, -14.28476303, ..., -14.08812374,
```

(continues on next page)

(continued from previous page)

```
        -15.06953462, -16.08068056],
       [-14.56935708, -14.61586487, -14.6496726 , ..., -14.51543286,
        -14.23547335, -13.93766339],
       [-15.34221339, -14.91692011, -14.4951453 , ...,  -0.78650147,
         -0.36136997,   0.08010293]])
Coordinates:
  * time         (time) datetime64[ns] 2000-01-...
  * realization  (realization) int64 0 1 2 ... 8 9
Attributes:
    units:          degC
    standard_name:  air_temperature
    long_name:      Mean air temperature at sur...
    title:          tas of member 01
```

3.2.2 Ensemble statistics

Beyond creating ensemble dataset the `xclim.ensembles` module contains functions for calculating statistics between realizations

Ensemble mean, standard-deviation, max & min

In the example below we use xclim's `ensemble_mean_std_max_min()` to calculate statistics across the 10 realizations in our test dataset. Output variables are created combining the original variable name `tas` with addtional ending indicating the statistic calculated on the realization dimension : `_mean`, `_stdev`, `_min`, `_max`

The resulting output now contains 4 derived variables from the original single variable in our ensemble dataset.

[5]:
```
ens_stats = ensembles.ensemble_mean_std_max_min(ens)
ens_stats
```

[5]:
```
<xarray.Dataset>
Dimensions:    (time: 1461)
Coordinates:
  * time       (time) datetime64[ns] 2000-01-01...
Data variables:
    tas_mean   (time) float64 -10.93 ... -10.27
    tas_stdev  (time) float64 4.376 4.26 ... 7.153
    tas_max    (time) float64 -5.458 ... 1.547
    tas_min    (time) float64 -16.51 ... -17.1
Attributes:
    history:  [2022-06-18 02:30:11] : Computati...
```

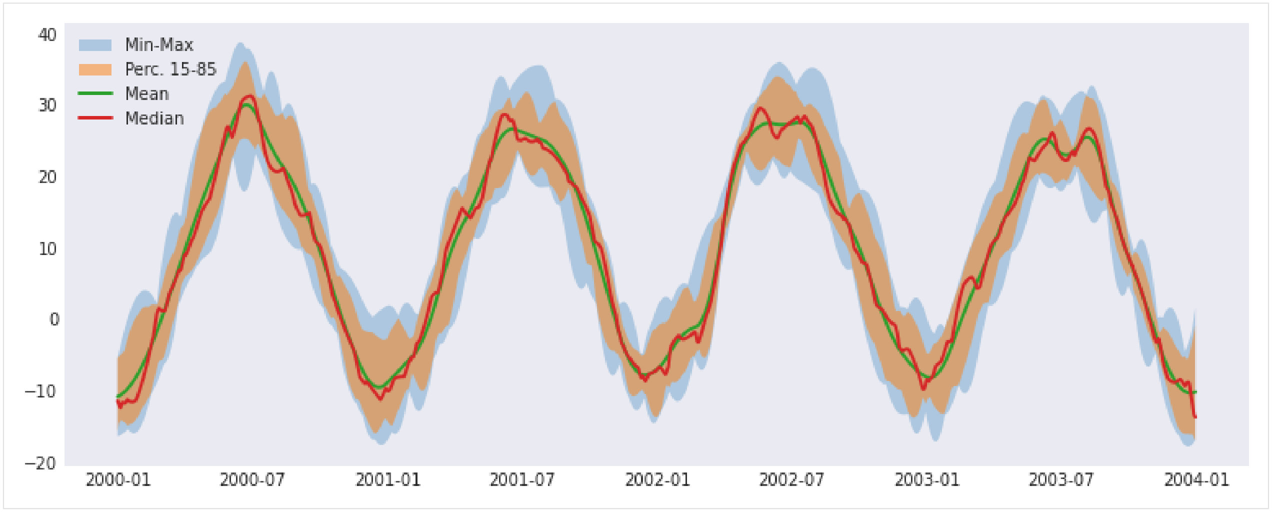
3.2.3 Ensemble percentiles

Here we use xclim's `ensemble_percentiles()` to calculate percentile values across the 10 realizations. The output has now a `percentiles` dimension instead of `realization`. Split variables can be created instead, by specifying `split=True` (the variable name `tas` will be appended with `_p{x}`). Compared to numpy's `percentile()` and xarray's `quantile()`, this method handles more efficiently dataset with invalid values and the chunking along the realization dimension (which is automatic when dask arrays are used).

[6]:
```
ens_perc = ensembles.ensemble_percentiles(ens, values=[15, 50, 85], split=False)
ens_perc
```

[6]:
```
<xarray.Dataset>
Dimensions:      (time: 1461, percentiles: 3)
Coordinates:
  * time         (time) datetime64[ns] 2000-01-...
  * percentiles  (percentiles) int64 15 50 85
Data variables:
    tas          (time, percentiles) float64 -1...
Attributes:
    units:          degC
    standard_name:  air_temperature
    long_name:      Mean air temperature at sur...
    title:          tas of member 01
    history:        [2022-06-18 02:30:11] : Com...
```

[7]:
```
fig, ax = plt.subplots()
ax.fill_between(
    ens_stats.time.values,
    ens_stats.tas_min,
    ens_stats.tas_max,
    alpha=0.3,
    label="Min-Max",
)
ax.fill_between(
    ens_perc.time.values,
    ens_perc.tas.sel(percentiles=15),
    ens_perc.tas.sel(percentiles=85),
    alpha=0.5,
    label="Perc. 15-85",
)
ax._get_lines.get_next_color()  # Hack to get different line
ax._get_lines.get_next_color()
ax.plot(ens_stats.time.values, ens_stats.tas_mean, linewidth=2, label="Mean")
ax.plot(
    ens_perc.time.values, ens_perc.tas.sel(percentiles=50), linewidth=2, label="Median"
)
ax.legend()
plt.show()
```



3.3 Ensemble-Reduction Techniques

`xclim.ensembles` provides means of reducing the number of candidates in a sample to get a reasonable and representative spread of outcomes using a reduced number of candidates. By reducing the number of realizations in a strategic manner, we can significantly reduce the number of realizations to examine, while maintaining statistical representation of original dataset. This is particularly useful when computation power or time is a factor.

For more information on the principles and methods behind ensemble reduction techniques, see: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0152495 and https://doi.org/10.1175/JCLI-D-14-00636.1

Selection Criteria

The following example considers 50 member ensemble with a total of 6 criteria considered (3 variable deltas * 2 time horizons). Our goal is to reduce this number to a more manageable size while preserving the range of uncertainty across our different criteria.

[2]:
```
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr

from xclim import ensembles

# Using an xarray dataset of our criteria
ds_crit
```

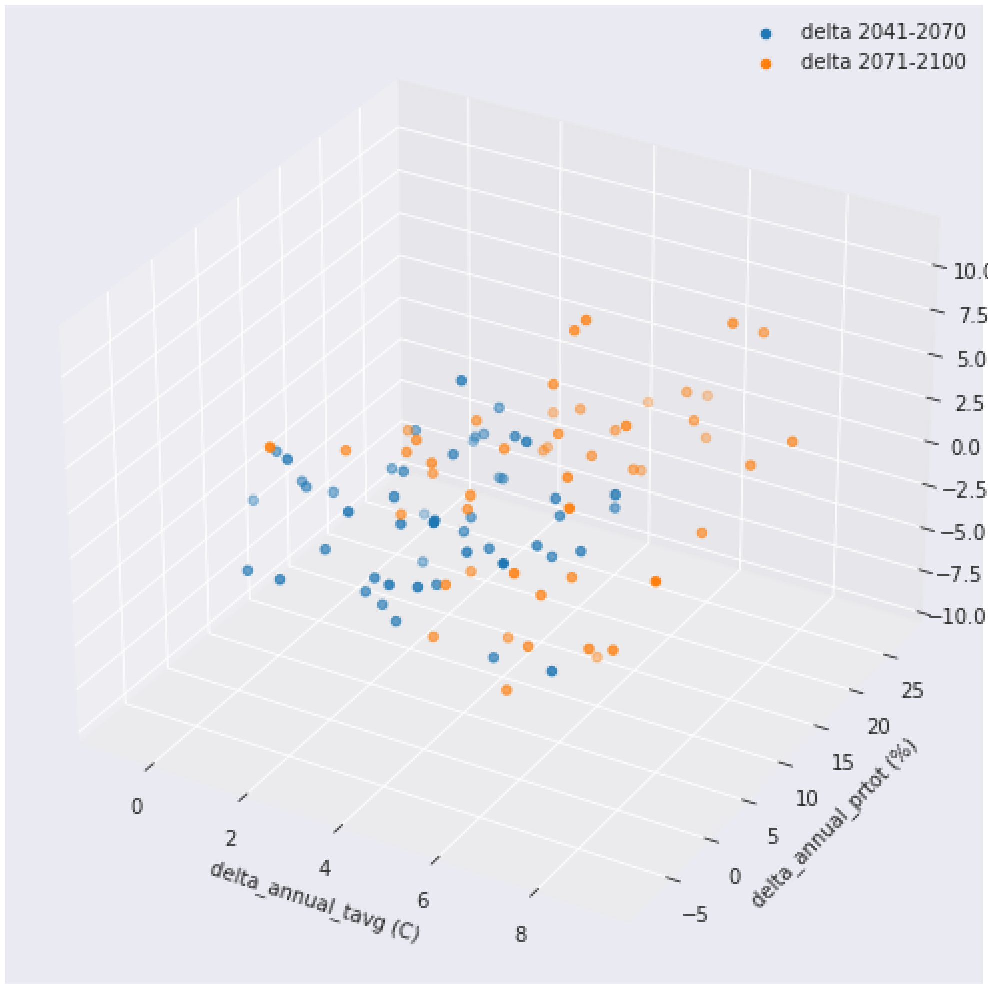
[2]:
```
<xarray.Dataset>
Dimensions:             (horizon: 2, realization: 50)
Coordinates:
  * horizon             (horizon) <U9 '2041-2070' '2071-2100'
Dimensions without coordinates: realization
Data variables:
    delta_annual_tavg   (horizon, realization) float64 5.646 3.6 ... 5.594 6.144
```

(continues on next page)

(continued from previous page)

```
    delta_annual_prtot  (horizon, realization) float64 14.42 -1.739 ... 20.69
    delta_JJA_prtot     (horizon, realization) float64 -1.108 -0.7181 ... 3.48
```

[3]:
```python
plt.style.use("seaborn-dark")
plt.rcParams["figure.figsize"] = (13, 5)
fig = plt.figure(figsize=(11, 9))
ax = plt.axes(projection="3d")

for h in ds_crit.horizon:
    ax.scatter(
        ds_crit.sel(horizon=h).delta_annual_tavg,
        ds_crit.sel(horizon=h).delta_annual_prtot,
        ds_crit.sel(horizon=h).delta_JJA_prtot,
        label=f"delta {h.values}",
    )

ax.set_xlabel("delta_annual_tavg (C)")
ax.set_ylabel("delta_annual_prtot (%)")
ax.set_zlabel("delta_JJA_prtot (%)")
plt.legend()
plt.show()
```



Ensemble reduction techniques in `xclim` require a 2D array with dimensions of `criteria` (values) and `realization` (runs/simulations).

```
[4]: # Create 2d xr.DataArray containing criteria values
crit = None
for h in ds_crit.horizon:
    for v in ds_crit.data_vars:
        if crit is None:
            crit = ds_crit[v].sel(horizon=h)
        else:
            crit = xr.concat((crit, ds_crit[v].sel(horizon=h)), dim="criteria")
crit.name = "criteria"
crit.shape
```

```
[4]: (6, 50)
```

3.3.1 K-Means reduce ensemble

The `kmeans_reduce_ensemble` works by grouping realizations into sub-groups based on the provided critera and retaining a representative `realization` per sub-group.

For a real-world example of the K-means clustering algorithm applied to climate data selection, see: https://doi.org/10.1371/journal.pone.0152495 and https://doi.org/10.1175/JCLI-D-11-00440.1

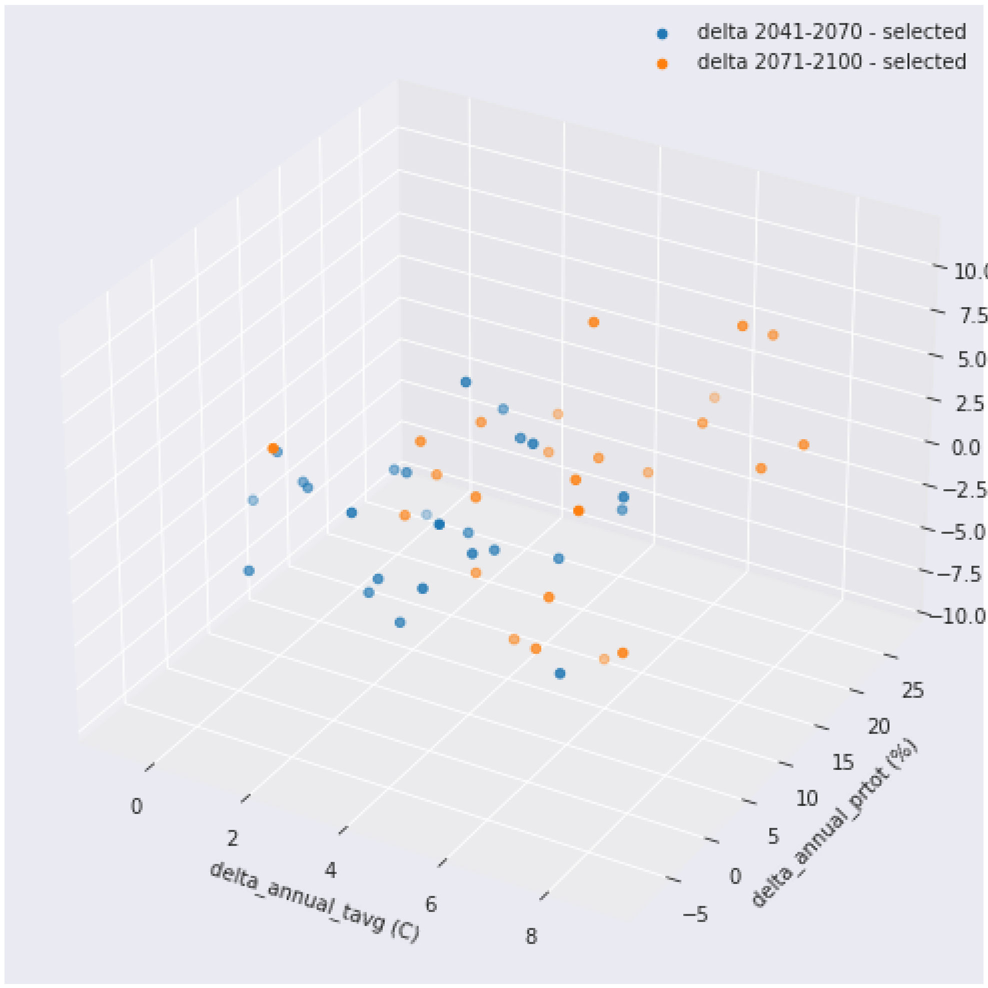
The following example uses `method = dict(n_clusters=25)` in order to take the original `50` realizations and reduce them down to `25`. The function itself returns the `ids` (indexes: `int`) of the realizations, which can then be used to select the data from the original ensemble.

[5]:
```
ids, cluster, fig_data = ensembles.kmeans_reduce_ensemble(
    data=crit, method={"n_clusters": 25}, random_state=42, make_graph=True
)
ds_crit.isel(realization=ids)
```

[5]:
```
<xarray.Dataset>
Dimensions:              (horizon: 2, realization: 25)
Coordinates:
  * horizon              (horizon) <U9 '2041-2070' '2071-2100'
Dimensions without coordinates: realization
Data variables:
    delta_annual_tavg    (horizon, realization) float64 5.646 4.468 ... 5.594
    delta_annual_prtot   (horizon, realization) float64 14.42 -1.352 ... 27.31
    delta_JJA_prtot      (horizon, realization) float64 -1.108 3.299 ... 0.4022
```

With this reduced number, we can now compare the distribution of the selection versus the original ensemble of simulations.

[6]:
```
plt.style.use("seaborn-dark")
fig = plt.figure(figsize=(11, 9))
ax = plt.axes(projection="3d")

for h in ds_crit.horizon:
    ax.scatter(
        ds_crit.sel(horizon=h, realization=ids).delta_annual_tavg,
        ds_crit.sel(horizon=h, realization=ids).delta_annual_prtot,
        ds_crit.sel(horizon=h, realization=ids).delta_JJA_prtot,
        label=f"delta {h.values} - selected",
    )

ax.set_xlabel("delta_annual_tavg (C)")
ax.set_ylabel("delta_annual_prtot (%)")
ax.set_zlabel("delta_JJA_prtot (%)")
plt.legend()
plt.show()
```

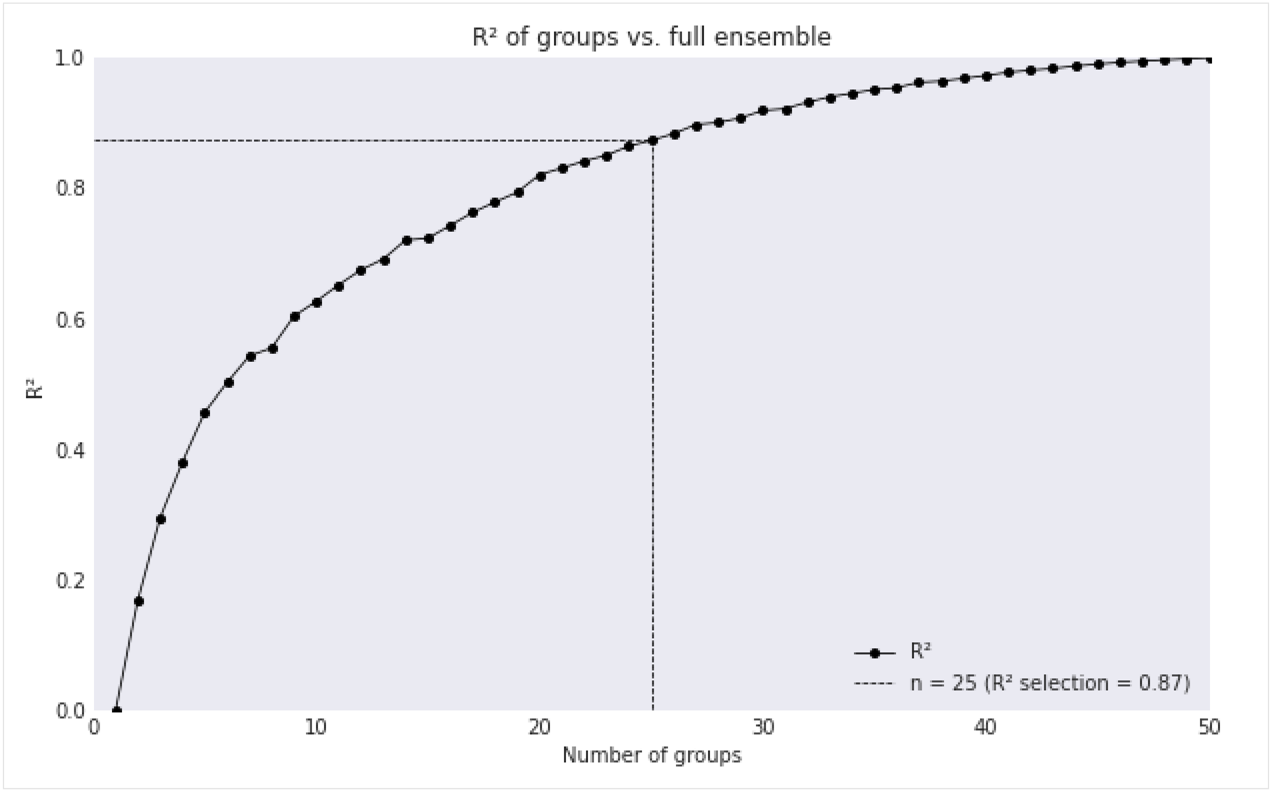


The function optionally produces a data dictionary for figure production of the associated R^2 profile.

The function `ensembles.plot_rsqprofile` provides plotting for evaluating the proportion of total variance in climate realizations that is covered by the selection.

In this case ~88% of the total variance in original ensemble is covered by the selection.

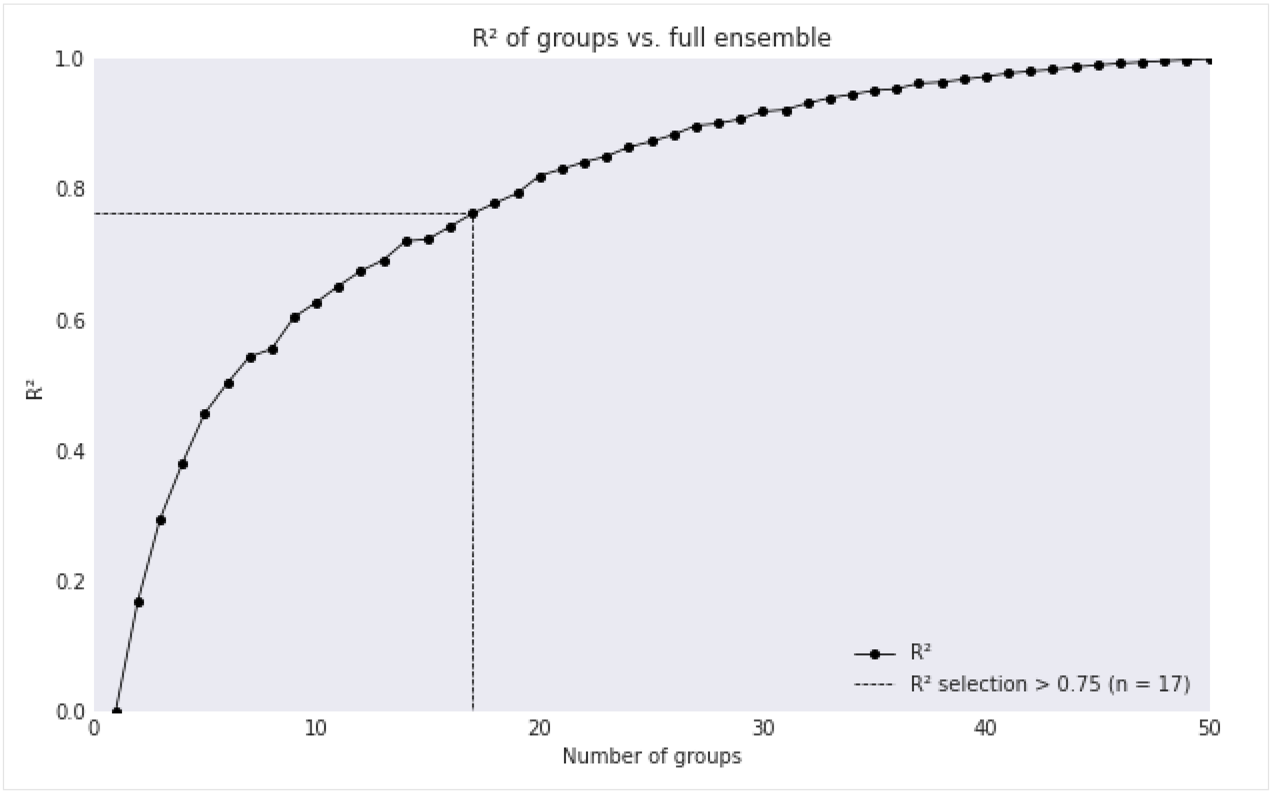
[7]:
```
ensembles.plot_rsqprofile(fig_data)
```



Alternatively we can use `method = {'rsq_cutoff':Float}` or `method = {'rsq_optimize':None}` * For example with `rsq_cutoff` we instead find the number of realizations needed to cover the provided R^2 value

```
[8]: ids1, cluster1, fig_data1 = ensembles.kmeans_reduce_ensemble(
         data=crit, method={"rsq_cutoff": 0.75}, random_state=42, make_graph=True
     )
     ensembles.plot_rsqprofile(fig_data1)
     ds_crit.isel(realization=ids1)
```

```
[8]: <xarray.Dataset>
     Dimensions:              (horizon: 2, realization: 17)
     Coordinates:
       * horizon              (horizon) <U9 '2041-2070' '2071-2100'
     Dimensions without coordinates: realization
     Data variables:
         delta_annual_tavg    (horizon, realization) float64 5.646 4.468 ... 6.144
         delta_annual_prtot   (horizon, realization) float64 14.42 -1.352 ... 20.69
         delta_JJA_prtot      (horizon, realization) float64 -1.108 3.299 ... 3.48
```



3.3.2 KKZ reduce ensemble

`xclim` also makes available a similar ensemble reduction algorithm, `ensembles.kkz_reduce_ensemble`. see: https://doi.org/10.1175/JCLI-D-14-00636.1

The advantage of this algorithm is largely that fewer realizations are needed in order to reach the same level of representative members than the K-means clustering algorithm, as the KKZ methods tends towards identifying members that fall towards the extremes of criteria values.

This technique also produces nested selection results, where additional increase in desired selection size does not alter the previous choices, which is not the case for the K-means algorithm.

[9]:
```
ids = ensembles.kkz_reduce_ensemble(crit, num_select=10)
ds_crit.isel(realization=ids)
```

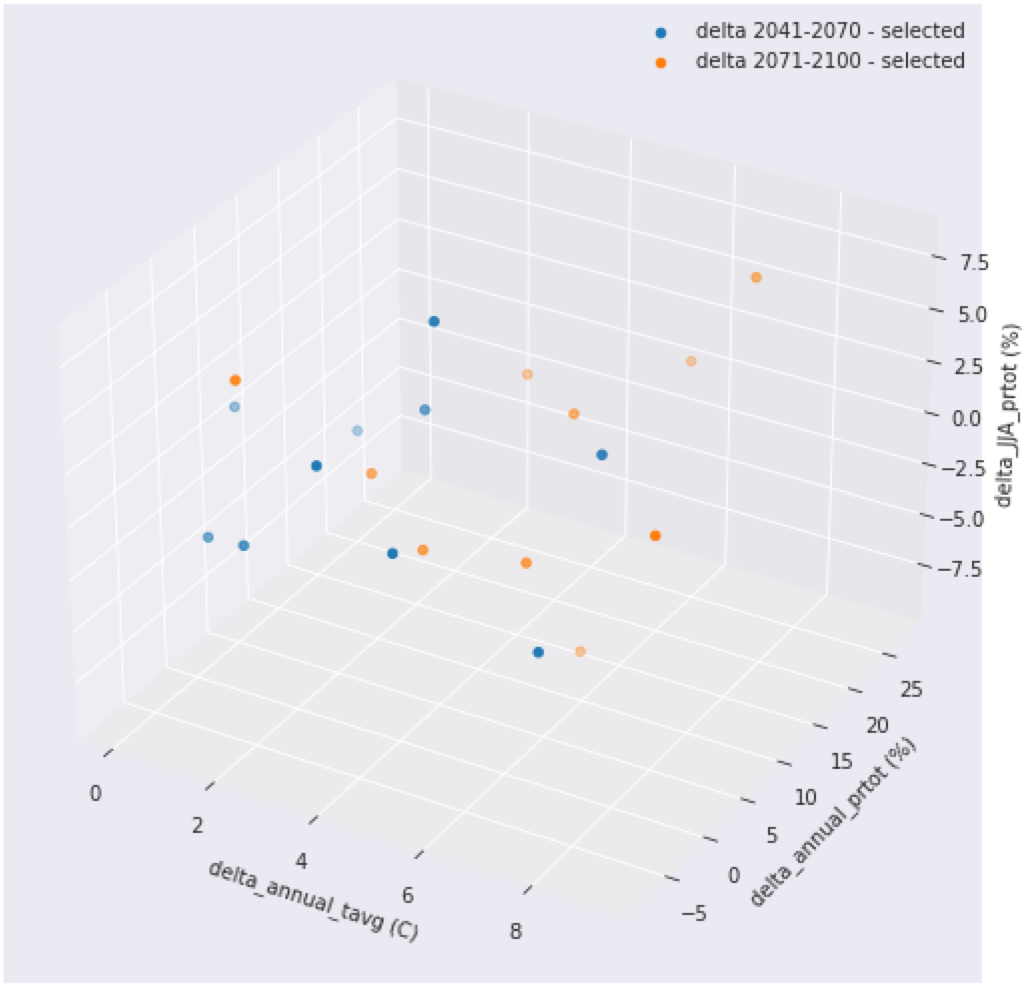
[9]:
```
<xarray.Dataset>
Dimensions:              (horizon: 2, realization: 10)
Coordinates:
  * horizon              (horizon) <U9 '2041-2070' '2071-2100'
Dimensions without coordinates: realization
Data variables:
    delta_annual_tavg    (horizon, realization) float64 1.719 6.405 ... 7.449
    delta_annual_prtot   (horizon, realization) float64 9.611 1.527 ... 22.34
    delta_JJA_prtot      (horizon, realization) float64 -0.1268 -4.622 ... 7.207
```

[10]:
```
plt.style.use("seaborn-dark")
fig = plt.figure(figsize=(9, 9))
```

(continues on next page)

(continued from previous page)

```
ax = plt.axes(projection="3d")

for h in ds_crit.horizon:

    ax.scatter(
        ds_crit.sel(horizon=h, realization=ids).delta_annual_tavg,
        ds_crit.sel(horizon=h, realization=ids).delta_annual_prtot,
        ds_crit.sel(horizon=h, realization=ids).delta_JJA_prtot,
        label=f"delta {h.values} - selected",
    )

ax.set_xlabel("delta_annual_tavg (C)")
ax.set_ylabel("delta_annual_prtot (%)")
ax.set_zlabel("delta_JJA_prtot (%)")
plt.legend()
plt.show()
```



3.3.3 KKZ algorithm vs K-Means algorithm

To give a better sense of the differences between **Nested (KKZ)** and **Unnested (K-Means)** results, we can progressively identify members that would be chosen by each algorithm through iterative fashion.

[11]:
```
## NESTED results using KKZ
for n in np.arange(5, 15, 3):
    ids = ensembles.kkz_reduce_ensemble(crit, num_select=n)
    print(ids)
```

```
[19, 24, 33, 3, 21]
[19, 24, 33, 3, 21, 18, 35, 48]
[19, 24, 33, 3, 21, 18, 35, 48, 40, 39, 29]
[19, 24, 33, 3, 21, 18, 35, 48, 40, 39, 29, 11, 2, 8]
```

[12]:
```
## UNNESTED results using k-means
for n in np.arange(5, 15, 3):
    ids, cluster, fig_data = ensembles.kmeans_reduce_ensemble(
        crit, method={"n_clusters": n}, random_state=42, make_graph=True
    )
    print(ids)
```

```
[7, 12, 27, 35, 45]
[7, 12, 19, 26, 27, 29, 36, 49]
[0, 10, 12, 14, 19, 32, 35, 38, 39, 45, 49]
[2, 12, 14, 16, 17, 19, 22, 27, 33, 39, 40, 45, 47, 49]
```

While the **Nested** feature of the KKZ results is typically advantageous, it can sometimes result in unbalanced coverage of the original ensemble. **In general careful consideration and validation of selection results is suggested when ``n`` is small, regardless of the technique used.**

To illustrate a simple example using only 2 of our criteria shows differences in results between the two techniques:

The **KKZ** algorithm iteratively maximizes distance from previous selected candidates - potentially resulting in 'off-center' results versus the original ensemble

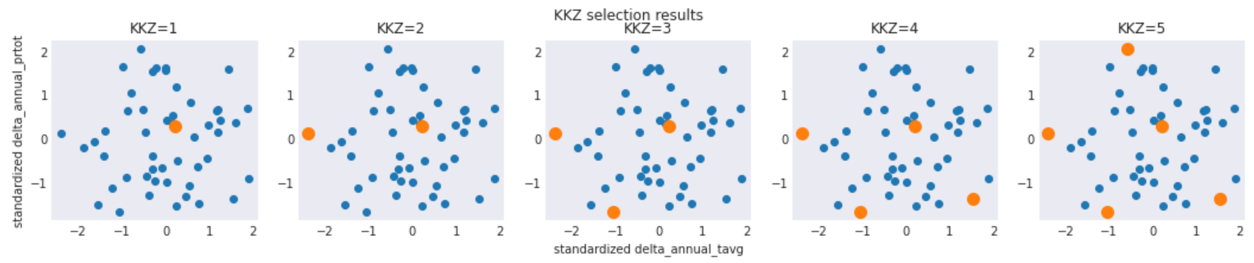
The **K-means** algorithm will redivide the data space with each iteration producing results that are consistently centered on the original ensemble but lacking coverage in the extremes

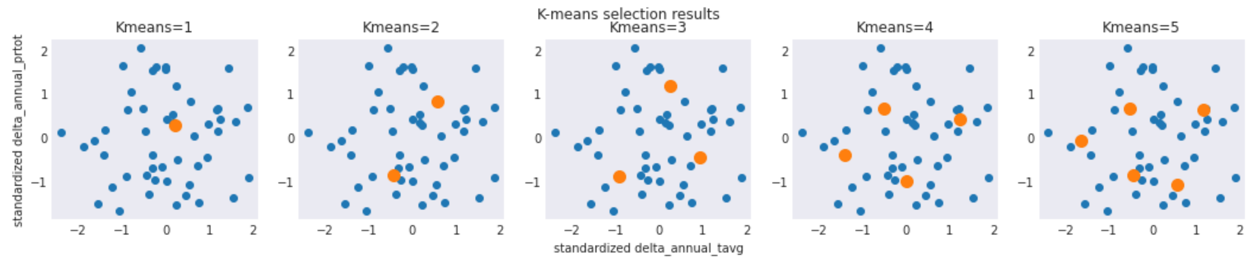
[13]:
```
df = crit.isel(criteria=[0, 1])

# Use standardized data in the plot so that selection distances is better visualized
df = (df - df.mean("realization")) / df.std("realization")

plt.figure(figsize=(18, 3))
for n in np.arange(1, 6):
    plt.subplot(1, 5, n, aspect="equal")
    plt.scatter(df.isel(criteria=0), df.isel(criteria=1))
    ids_KKZ = ensembles.kkz_reduce_ensemble(crit.isel(criteria=[0, 1]), num_select=n)
    plt.scatter(
        df.isel(criteria=0, realization=ids_KKZ),
        df.isel(criteria=1, realization=ids_KKZ),
        s=100,
    )
    plt.title(f"KKZ={n}")
```

(continues on next page)

(continued from previous page)

```
    if n == 1:
        plt.ylabel("standardized delta_annual_prtot")
    if n == 3:
        plt.xlabel("standardized delta_annual_tavg")
plt.suptitle("KKZ selection results")

plt.figure(figsize=(18, 3))
for n in np.arange(1, 6):
    plt.subplot(1, 5, n, aspect="equal")
    plt.scatter(df.isel(criteria=0), df.isel(criteria=1))
    ids_Kmeans, c, figdata = ensembles.kmeans_reduce_ensemble(
        crit.isel(criteria=[0, 1]),
        method={"n_clusters": n},
        random_state=42,
        make_graph=True,
    )
    plt.scatter(
        df.isel(criteria=0, realization=ids_Kmeans),
        df.isel(criteria=1, realization=ids_Kmeans),
        s=100,
    )
    plt.title(f"Kmeans={n}")
    if n == 1:
        plt.ylabel("standardized delta_annual_prtot")
    if n == 3:
        plt.xlabel("standardized delta_annual_tavg")
plt.suptitle("K-means selection results")
plt.show()
```





```
[ ]:
```

3.4 Frequency analysis

Frequency analysis refers to the study of the probability of occurrence of events. It's often used in regulatory contexts to determine design values for infrastructures. For example, your city might require that road drainage systems be able to cope with a level of rainfall that is exceeded only once every 20 years on average. This 20-year return event, the infrastructure *design value*, is computed by first extracting precipitation annual maxima from a rainfall observation time series, fitting a statistical distribution to the maxima, then estimating the 95th percentile (1:20 chance of being exceeded).

To facilitate this type of analysis on a large number of time series from model simulations or observations, `xclim` packs a few common utility functions. In the following example, we're estimating the 95th percentile of the daily precipitation maximum over the May-October period using a Generalized Extreme Value distribution.

Note that at the moment, all frequency analysis functions are hard-coded to operate along the `time` dimension.

Let's first create a synthetic time series of daily precipitation.

[1]:
```python
from __future__ import annotations

import warnings

import numpy as np
import xarray as xr

warnings.simplefilter("ignore")
from scipy.stats import bernoulli, gamma

from xclim.core.missing import missing_pct
from xclim.indices.generic import select_resample_op
from xclim.indices.stats import fa, fit, frequency_analysis, parametric_quantile

# Create synthetic daily precipitation time series (mm/d)
n = 50 * 366
start = np.datetime64("1950-01-01")
time = start + np.timedelta64(1, "D") * range(n)
# time = xr.cftime_range(start="1950-01-01", periods=n)

# Generate wet (1) /dry (0) days, then multiply by rain magnitude.
wet = bernoulli.rvs(0.1, size=n)
intensity = gamma(a=4, loc=1, scale=6).rvs(n)
pr = xr.DataArray(
    wet * intensity,
    dims=("time",),
    coords={"time": time},
    attrs={"units": "mm/d", "standard_name": "precipitation_flux"},
)
pr
```

[1]:
```
<xarray.DataArray (time: 18300)>
array([0., 0., 0., ..., 0., 0., 0.])
Coordinates:
  * time     (time) datetime64[ns] 1950-01-01 1950-01-02 ... 2000-02-07
Attributes:
```

(continues on next page)

(continued from previous page)

```
    units:           mm/d
    standard_name:   precipitation_flux
```

The `frequency_analysis` function combines all the necessary steps to estimate our design value:

1. Extract May to October period (`month=[5,6,7,8,9,10]`)
2. Extract maxima (`mode="max"`)
3. Fit the GEV distribution on the maxima (`dist="genextreme"`)
4. Compute the value exceeded, on average, once every 20 years (`t=20`)

Note that `xclim` essentially wraps `scipy.stats` distributions, so many distributions like `norm`, `gumbel_r`, `lognorm`, etc. are supported.

[2]:
```
# Compute the design value
frequency_analysis(
    pr, t=20, dist="genextreme", mode="max", freq="Y", month=[5, 6, 7, 8, 9, 10]
)
```

[2]:
```
<xarray.DataArray (return_period: 1)>
array([74.69670501])
Coordinates:
  * return_period  (return_period) int64 20
Attributes:
    units:           mm/d
    standard_name:   precipitation_flux
    long_name:       genextreme quantiles
    description:     Quantiles estimated by the genextreme distribution
    method:          ML
    estimator:       Maximum likelihood
    scipy_dist:      genextreme
    history:         [2022-06-18 02:31:58] fit: Estimate distribution paramete...
    cell_methods:    dparams: ppf
    mode:            max
```

In practice, it's often useful to be able to save intermediate results, for example the parameters of the fitted distribution, so in the following we crack open what goes on behind the `frequency_analysis` function.

The first step of the frequency analysis is to extract the May-October maxima. This is done using the `indices.generic.select_resample_op` function, which applies an operator (`op`) on a resampled time series. It can also select portion of the year, such as climatological seasons (e.g. 'DJF' for winter months), or individual months (e.g. `month=[1]` for January).

[3]:
```
sub = select_resample_op(pr, op="max", freq="Y", month=[5, 6, 7, 8, 9, 10])
sub
```

[3]:
```
<xarray.DataArray (time: 51)>
array([45.51699627, 42.69503185, 93.77099883, 71.57063023, 43.7633359 ,
       40.57953409, 59.96725296, 35.97142182, 44.21731934, 43.22580351,
       52.09874996, 46.30095552, 36.0171995 , 80.43468681, 53.87478782,
       35.71667243, 58.28094762, 57.21638637, 61.2696387 , 39.57774575,
       46.45441392, 41.03457489, 61.64630529, 55.93411389, 64.01272705,
       55.94322184, 54.03169748, 54.00135074, 37.2956566 , 34.58466141,
       57.74490691, 50.66924443, 41.02187039, 59.96527618, 43.01162305,
```

(continues on next page)

(continued from previous page)

```
       42.03537549, 43.52530979, 52.05684448, 52.36678734, 65.6718084 ,
       50.75366261, 73.07024441, 61.86338567, 56.74860485, 50.73516572,
       43.47039131, 46.80618132, 44.29597376, 39.1895819 , 60.79557524,
               nan])
Coordinates:
  * time     (time) datetime64[ns] 1950-12-31 1951-12-31 ... 2000-12-31
Attributes:
    units:          mm/d
    standard_name:  precipitation_flux
```

The next step is to fit the statistical distribution on these maxima. This is done by the `fit` method, which takes as argument the sample series, the distribution's name and the parameter estimation method. The fit is done by default using the Maximum Likelihood algorithm. For some extreme value distributions however, the maximum likelihood is not always robust, and `xclim` offers the possibility to use Probability Weighted Moments (PWM) to estimate parameters. Note that the `lmoments3` package which is used by `xclim` to compute the PWM only supports `expon`, `gamma`, `genextreme`, `genpareto`, `gumbel_r`, `pearson3` and `weibull_min`.

[4]:
```
# The fitting dimension is hard-coded as `time`.
params = fit(sub, dist="genextreme")
params
```

[4]:
```
<xarray.DataArray (dparams: 3)>
array([-0.05436137, 46.01117703,  8.8990726 ])
Coordinates:
  * dparams  (dparams) <U5 'c' 'loc' 'scale'
Attributes:
    original_units:          mm/d
    original_standard_name:  precipitation_flux
    long_name:               genextreme parameters
    description:             Parameters of the genextreme distribution
    method:                  ML
    estimator:               Maximum likelihood
    scipy_dist:              genextreme
    units:
    history:                 [2022-06-18 02:31:58] fit: Estimate distribution...
```

Finally, the last step is to compute the percentile, or quantile, using the fitted parameters, using the `parametric_quantile` function. The function uses metadata stored in attributes of the parameters generated by `fit` to determine what distribution to use and what are the units of the quantiles. Here we need to pass the quantile (values between 0 and 1), which for exceedance probabilities is just :math`1 - 1/T`.

[5]:
```
parametric_quantile(params, q=1 - 1.0 / 20)
```

[5]:
```
<xarray.DataArray (quantile: 1)>
array([74.69670501])
Coordinates:
  * quantile  (quantile) float64 0.95
Attributes:
    units:          mm/d
    standard_name:  precipitation_flux
    long_name:      genextreme quantiles
    description:    Quantiles estimated by the genextreme distribution
```

(continues on next page)

(continued from previous page)

```
    method:          ML
    estimator:       Maximum likelihood
    scipy_dist:      genextreme
    history:         [2022-06-18 02:31:58] fit: Estimate distribution paramete...
    cell_methods:    dparams: ppf
```

As a convenience utility, the two last steps (`fit` and `parametric_quantile`) are bundled into the `fa` function, which takes care of converting the return period into a quantile value, and renames the `quantile` output dimension to `return_period`. This dimension renaming is done to avoid name clashes with the `quantile` method. Indeed, it's often necessary when analysing large ensembles, or probabilistic samples, to compute the quantiles of the quantiles, which will cause `xarray` to raise an error. The `mode` argument specifies whether we are working with maxima (max) or minima (min). This is important because a 100-year return period value for minima corresponds to a 0.01 quantile, while a 100-year return period value for maxima corresponds to a 0.99 quantile.

[6]:
```
fa(sub, t=20, dist="genextreme", mode="max")
```

[6]:
```
<xarray.DataArray (return_period: 1)>
array([74.69670501])
Coordinates:
  * return_period  (return_period) int64 20
Attributes:
    units:           mm/d
    standard_name:   precipitation_flux
    long_name:       genextreme quantiles
    description:     Quantiles estimated by the genextreme distribution
    method:          ML
    estimator:       Maximum likelihood
    scipy_dist:      genextreme
    history:         [2022-06-18 02:31:58] fit: Estimate distribution paramete...
    cell_methods:    dparams: ppf
    mode:            max
```

3.4.1 Handling missing values

When working with observations from weather stations, there are often stretches of days without measurements due to equipment malfunction. Practitioners usually do not want to ignore entire years of data due to a few missing days, so one option is to record annual maxima only if there are no more than a given percentage of missing values, say 5%. These kinds of filters can easily be applied using `xclim`.

[7]:
```
# Set the first half of the first year as missing.
pr[:200] = np.nan

# Compute vector returning which years should be considered missing.
null = missing_pct(pr, tolerance=0.05, freq="Y", month=[5, 6, 7, 8, 9, 10])

# Compute stats on masked values
fa(sub.where(~null), t=20, dist="genextreme", mode="high")
```

[7]:
```
<xarray.DataArray (return_period: 1)>
array([74.96847452])
Coordinates:
```

(continues on next page)

(continued from previous page)

```
  * return_period  (return_period) int64 20
Attributes:
    units:          mm/d
    standard_name:  precipitation_flux
    long_name:      genextreme quantiles
    description:    Quantiles estimated by the genextreme distribution
    method:         ML
    estimator:      Maximum likelihood
    scipy_dist:     genextreme
    history:        [2022-06-18 02:31:59] fit: Estimate distribution paramete...
    cell_methods:   dparams: ppf
    mode:           high
```

3.5 Customizing and controlling xclim

xclim's behaviour can be controlled globally or contextually through `xclim.set_options`, which acts the same way as `xarray.set_options`. For the extension of xclim with the addition of indicators, see the *Extending xclim* notebook.

[1]:
```
from __future__ import annotations

import xarray as xr

import xclim
from xclim.testing import open_dataset
```

Let's create fake data with some missing values and mask every 10th, 20th and 30th of the month.This represents 9.6-10% of masked data for all months except February where it is 7.1%.

[2]:
```
tasmax = (
    xr.tutorial.open_dataset("air_temperature")
    .air.resample(time="D")
    .max(keep_attrs=True)
)
tasmax = tasmax.where(tasmax.time.dt.day % 10 != 0)
```

3.5.1 Checks

Above, we created fake temperature data from a xarray tutorial dataset that doesn't have all the standard CF attributes. By default, when triggering a computation with an Indicator from xclim, warnings will be raised:

[3]:
```
tx_mean = xclim.atmos.tx_mean(tasmax=tasmax, freq="MS")  # compute monthly max tasmax
```

```
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/core/cfchecks.py:41: UserWarning: Variable does not have a `cell_
↪methods` attribute.
  _check_cell_methods(
```

(continues on next page)

(continued from previous page)

```
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/core/cfchecks.py:45: UserWarning: Variable does not have a `standard_
↪name` attribute.
  check_valid(vardata, "standard_name", data["standard_name"])
```

Setting `cf_compliance` to `'log'` mutes those warnings and sends them to the log instead.

```
[4]: xclim.set_options(cf_compliance="log")

     tx_mean = xclim.atmos.tx_mean(tasmax=tasmax, freq="MS")  # compute monthly max tasmax
```

3.5.2 Adding translated metadata

With the help of its internationalization module (`xclim.core.locales`), xclim can add translated metadata to the output of the indicators. The metadata is *not* translated on-the-fly, but translations are manually written for each indicator and metadata field. Currently, all indicators have a french translation, but users can add more choices. See *Internationalization* and *Extending xclim*.

In the example below, notice the added `long_name_fr` and `description_fr` attributes. Also, the use of `set_options` as a context makes this configuration transient, only valid within the context.

```
[5]: with xclim.set_options(metadata_locales=["fr"]):
         out = xclim.atmos.tx_max(tasmax=tasmax)
     out.attrs
```

```
[5]: {'units': 'K',
      'cell_methods': ' time: maximum over days',
      'history': "[2022-06-18 02:29:54] tx_max: TX_MAX(tasmax=air, freq='YS') with options␣
     ↪check_missing=any - xclim version: 0.37.0",
      'standard_name': 'air_temperature',
      'long_name': 'Maximum daily maximum temperature',
      'description': 'Annual maximum of daily maximum temperature.',
      'long_name_fr': 'Maximum de la température journalière',
      'description_fr': 'Maximum annuel de la température journalière maximale.'}
```

3.5.3 Missing values

One can also globally change the missing method.

Change the default missing method to "pct" and set its tolerance to 8%:

```
[6]: xclim.set_options(check_missing="pct", missing_options={"pct": {"tolerance": 0.08}})

     tx_mean = xclim.atmos.tx_mean(tasmax=tasmax, freq="MS")  # compute monthly max tasmax
     tx_mean.sel(time="2013", lat=75, lon=200)
```

```
[6]: <xarray.DataArray 'tx_mean' (time: 12)>
     array([      nan, 242.76694,       nan,       nan,       nan,       nan,
                  nan,       nan,       nan,       nan,       nan,       nan],
           dtype=float32)
     Coordinates:
       * time     (time) datetime64[ns] 2013-01-01 2013-02-01 ... 2013-12-01
```

(continues on next page)

(continued from previous page)

```
    lat      float32 75.0
    lon      float32 200.0
Attributes:
    units:          K
    cell_methods:    time: mean over days
    history:        [2022-06-18 02:29:54] tx_mean: TX_MEAN(tasmax=air, freq='...
    standard_name:  air_temperature
    long_name:      Mean daily maximum temperature
    description:    Monthly mean of daily maximum temperature.
```

Only February has non-masked data. Let's say we want to use the "wmo" method (and its default options), but only once, we can do:

```
[7]: with xclim.set_options(check_missing="wmo"):
         tx_mean = xclim.atmos.tx_mean(
             tasmax=tasmax, freq="MS"
         )  # compute monthly max tasmax
     tx_mean.sel(time="2013", lat=75, lon=200)
```

```
[7]: <xarray.DataArray 'tx_mean' (time: 12)>
     array([246.4122 , 242.76694, 250.18001, 260.53598, 268.20145, 274.92004,
            277.01144, 273.31146, 270.30484, 263.94357, 254.68298, 251.45862],
           dtype=float32)
     Coordinates:
       * time     (time) datetime64[ns] 2013-01-01 2013-02-01 ... 2013-12-01
         lat      float32 75.0
         lon      float32 200.0
     Attributes:
         units:          K
         cell_methods:    time: mean over days
         history:        [2022-06-18 02:29:54] tx_mean: TX_MEAN(tasmax=air, freq='...
         standard_name:  air_temperature
         long_name:      Mean daily maximum temperature
         description:    Monthly mean of daily maximum temperature.
```

This method checks that there is less than `nm=5` invalid values in a month and that there are no consecutive runs of `nc>=4` invalid values. Thus, every month is now valid.

Finally, it is possible for advanced users to register their own method. Xclim's missing methods are in fact based on class instances. Thus, to create a custom missing class, one should implement a subclass based on `xclim.core.checks.MissingBase` and overriding at least the `is_missing` method. The method should take a `null` argument and a `count` argument.

- `null` is a `DataArrayResample` instance of the resampled mask of invalid values in the input dataarray.
- `count` is the number of days in each resampled periods and any number of other keyword arguments.

The `is_missing` method should return a boolean mask, at the same frequency as the indicator output (same as `count`), where True values are for elements that are considered missing and masked on the output.

When registering the class with the `xclim.core.checks.register_missing_method` decorator, the keyword arguments will be registered as options for the missing method. One can also implement a `validate` static method that receives only those options and returns whether they should be considered valid or not.

[8]:
```
from xclim.core.missing import MissingBase, register_missing_method
from xclim.indices.run_length import longest_run


@register_missing_method("consecutive")
class MissingConsecutive(MissingBase):
    """Any period with more than max_n consecutive missing values is considered invalid""
↪"

    def is_missing(self, null, count, max_n=5):
        return null.map(longest_run, dim="time") >= max_n

    @staticmethod
    def validate(max_n):
        return max_n > 0
```

The new method is now accessible and usable with:

[9]:
```
with xclim.set_options(
    check_missing="consecutive", missing_options={"consecutive": {"max_n": 2}}
):
    tx_mean = xclim.atmos.tx_mean(
        tasmax=tasmax, freq="MS"
    )  # compute monthly max tasmax
tx_mean.sel(time="2013", lat=75, lon=200)
```

[9]:
```
<xarray.DataArray 'tx_mean' (time: 12)>
array([246.4122 , 242.76694, 250.18001, 260.53598, 268.20145, 274.92004,
       277.01144, 273.31146, 270.30484, 263.94357, 254.68298, 251.45862],
      dtype=float32)
Coordinates:
  * time     (time) datetime64[ns] 2013-01-01 2013-02-01 ... 2013-12-01
    lat      float32 75.0
    lon      float32 200.0
Attributes:
    units:          K
    cell_methods:    time: mean over days
    history:        [2022-06-18 02:29:58] tx_mean: TX_MEAN(tasmax=air, freq='...
    standard_name:  air_temperature
    long_name:      Mean daily maximum temperature
    description:    Monthly mean of daily maximum temperature.
```

3.6 Extending xclim

xclim tries to make it easy for users to add their own indices and indicators. The following goes into details on how to create *indices* and document them so that xclim can parse most of the metadata directly. We then explain the multiple ways new *Indicators* can be created and, finally, how we can regroup and structure them in virtual submodules.

Central to xclim are the **Indicators**, objects computating indices over climate variables, but xclim also provides other modules:

Where `subset` is a phantom module, kept for legacy code, as it only redirects the calls to `clisops.core.subset`.

This introduction will focus on the Indicator/Indice part of xclim and how one can extend it by implementing new ones.

3.6.1 Indices vs Indicators

Internally and in the documentation, xclim makes a distinction between “indices” and “indicators”.

indice

- A python function accepting DataArrays and other parameters (usually bultin types)
- Returns one or several DataArrays.
- Handles the units : checks input units and set proper CF-compliant output units. But doesn’t usually prescribe specific units, the output will at minimum have the proper dimensionality.
- Performs **no** other checks or set any (non-unit) metadata.
- Accessible through *xclim.indices*.

indicator

- An instance of a subclass of `xclim.core.indicator.Indicator` that wraps around an `indice` (stored in its `compute` property).
- Returns one or several DataArrays.
- Handles missing values, performs input data and metadata checks (see *usage*).
- Always ouputs data in the same units.
- Adds dynamically generated metadata to the output after computation.
- Accessible through *xclim.indicators*

Most metadata stored in the Indicators is parsed from the underlying indice documentation, so defining indices with complete documentation and an appropriate signature helps the process. The two next sections go into details on the definition of both objects.

Call sequence

The following graph shows the steps done when calling an Indicator. Attributes and methods of the Indicator object relating to those steps are listed on the right side.

3.6.2 Defining new indices

The annotated example below shows the general template to be followed when defining proper *indices*. In the comments `Ind` is the indicator instance that would be created from this function.

Note that it is not *needed* to follow these standards when writing indices that will be wrapped in indicators. Problems in parsing will not raise errors at runtime, but might raise warnings and will result in Indicators with poorer metadata than expected by most users, especially those that dynamically use indicators in other applications where the code is inaccessible, like web services.

The following code is another example.

[1]:
```
from __future__ import annotations

import xarray as xr

import xclim as xc
from xclim.core.units import convert_units_to, declare_units
from xclim.indices.generic import threshold_count


@declare_units(tasmax="[temperature]", thresh="[temperature]")
def tx_days_compare(
    tasmax: xr.DataArray, thresh: str = "0 degC", op: str = ">", freq: str = "YS"
):
    r"""Number of days where maximum daily temperature. is above or under a threshold.

    The daily maximum temperature is compared to a threshold using a given operator and␣
↪the number
    of days where the condition is true is returned.

    It assumes a daily input.

    Parameters
    ----------
    tasmax : xarray.DataArray
      Maximum daily temperature.
    thresh : str
      Threshold temperature to compare to.
    op : {'>', '<'}
      The operator to use.
      # A fixed set of choices can be imposed. Only strings, numbers, booleans or None␣
↪are accepted.
    freq : str
      Resampling frequency.

    Returns
    -------
    xarray.DataArray, [temperature]
      Maximum value of daily maximum temperature.
```

(continues on next page)

(continued from previous page)

```
    Notes
    -----
    Let :math:`TX_{ij}` be the maximum temperature at day :math:`i` of period :math:`j`.␣
↪Then the maximum
    daily maximum temperature for period :math:`j` is:

    .. math::

        TXx_j = max(TX_{ij})

    References
    ----------
    Smith, John and Tremblay, Robert, An dummy citation for examples in documentation. J.
↪ RTD. (2020).
    """
    thresh = convert_units_to(thresh, tasmax)
    out = threshold_count(tasmax, op, thresh, freq)
    out.attrs["units"] = "days"
    return out
```

Naming and conventions

Variable names should correspond to CMIP6 variables, whenever possible. The file `xclim/data/variables.yml` lists all variables that xclim can use when generating indicators from yaml files (see below), and new indices should try to reflect these also. For new variables, the `xclim.testing.get_all_CMIP6_variables` function downloads the official table of CMIP6 variables and puts everything in a dictionary. If possible, use variables names from this list, add them to `variables.yml` as needed.

Generic functions for common operations

The *xclim.indices.generic* submodule contains useful functions for common computations (like `threshold_count` or `select_resample_op`) and many basic indice functions, as defined by clix-meta. In order to reduce duplicate code, their use is recommended for xclim's indices. As previously said, the units handling has to be made explicitly when non trivial, *xclim.core.units* also exposes a few helpers for that (like `convert_units_to`, `to_agg_units` or `rate2amount`).

3.6.3 Defining new indicators

xclim's Indicators are instances of (subclasses of) `xclim.core.indicator.Indicator`. While they are the central to xclim, their construction can be somewhat tricky as a lot happens backstage. Essentially, they act as self-aware functions, taking a set of input variables (DataArrays) and parameters (usually strings, integers or floats), performing some health checks on them and returning one or multiple DataArrays, with CF-compliant (and potentially translated) metadata attributes, masked according to a given missing value set of rules. They define the following key attributes:

- the `identifier`, as string that uniquely identifies the indicator, usually all caps.
- the `realm`, one of "atmos", "land", "seaIce" or "ocean", classifying the domain of use of the indicator.
- the `compute` function that returns one or more DataArrays, the "indice",
- the `cfcheck` and `datacheck` methods that make sure the inputs are appropriate and valid.

- the `missing` function that masks elements based on null values in the input.
- all metadata attributes that will be attributed to the output and that document the indicator:
  - Indicator-level attribute are : `title`, `abstract`, `keywords`, `references` and `notes`.
  - Ouput variables attributes (respecting CF conventions) are: `var_name`, `standard_name`, `long_name`, `units`, `cell_methods`, `description` and `comment`.

Output variables attributes are regrouped in `Indicator.cf_attrs` and input parameters are documented in `Indicator.parameters`.

A particularity of Indicators is that each instance corresponds to a single class: when creating a new indicator, a new class is automatically created. This is done for easy construction of indicators based on others, like shown further down.

See the *class documentation* for more info on the meaning of each attribute. The indicators module contains over 50 examples of indicators to draw inspiration from.

Identifier vs python name

An indicator's identifier is **not** the same as the name it has within the python module. For example, `xc.atmos.relative_humidity` has `hurs` as its identifier. As explained below, indicator *classes* can be accessed through `xc.core.indicator.registry` with their *identifier*.

Metadata parsing vs explicit setting

As explained above, most metadata can be parsed from the indice's signature and docstring. Otherwise, it can always be set when creating a new Indicator instance *or* a new subclass. When *creating* an indicator, output metadata attributes can be given as strings, or list of strings in the case of indicator returning multiple outputs. However, they are stored in the `cf_attrs` list of dictionaries on the instance.

Internationalization of metadata

xclim offers the possibility to translate the main Indicator metadata field and automatically add the translations to the outputs. The mechnanic is explained in the *Internationalization* page.

Inputs and checks

xclim decides which input arguments of the indicator's call function are considered *variables* and which are *parameters* using the annotations of the underlying indice (the `compute` method). Arguments annotated with the `xarray.DataArray` type are considered *variables* and can be read from the dataset passed in `ds`.

Indicator creation

There a two ways for creating indicators:

1) By initializing an existing indicator (sub)class
2) From a dictionary

The first method is best when defining indicators in scripts of external modules and are explained here. The second is best used when building virtual modules through YAML files, and is explained further down and in the *submodule doc*.

Creating a new indicator that simply modifies a few metadata output of an existing one is a simple call like:

```
[2]: from xclim.core.indicator import registry
     from xclim.core.utils import wrapped_partial

     # An indicator based on tg_mean, but returning Celsius and fixed on annual resampling
     tg_mean_c = registry["TG_MEAN"](
         identifier="tg_mean_c",
         units="degC",
         title="Mean daily mean temperature but in degC",
         parameters=dict(freq="YS"),  # We inject the freq arg.
     )
```

```
[3]: print(tg_mean_c.__doc__)
```

```
Mean daily mean temperature but in degC (realm: atmos)

Resample the original daily mean temperature series by taking the mean over each period.

This indicator will check for missing values according to the method "from_context".
Based on indice :py:func:`~xclim.indices._simple.tg_mean`.
With injected parameters: freq=YS.

Parameters
----------
tas : str or DataArray
  Mean daily temperature.
  Default : `ds.tas`. [Required units : [temperature]]
ds : Dataset, optional
  A dataset with the variables given by name.
  Default : None.
indexer :
  Indexing parameters to compute the indicator on a temporal subset of the data. It␣
↪accepts the same arguments as :py:func:`xclim.indices.generic.select_time`.
  Default : None.

Returns
-------
tg_mean : DataArray
  Mean daily mean temperature (air_temperature) [K]
  cell_methods: time: mean over days
  description: {freq} mean of daily mean temperature.

Notes
-----
Let :math:`TN_i` be the mean daily temperature of day :math:`i`, then for a period :math:
↪`p` starting at
day :math:`a` and finishing on day :math:`b`:

.. math::

   TG_p = \frac{\sum_{i=a}^{b} TN_i}{b - a + 1}
```

The registry is a dictionary mapping indicator identifiers (in uppercase) to their class. This way, we could subclass `tg_mean` to create our new indicator. `tg_mean_c` is the exact same as `atmos.tg_mean`, but outputs the result in Celsius instead of Kelvins, has a different title and removes control over the `freq` argument, resampling to "YS". The `identifier` keyword is here needed in order to differentiate the new indicator from `tg_mean` itself. If it wasn't given, a warning would have been raised and further subclassing of `tg_mean` would have in fact subclassed `tg_mean_c`, which is not wanted!

By default, indicator classes are registered in `xclim.core.indicator.registry`, using their identifier which is prepended by the indicator's module **if** that indicator is declared outisde xclim. An "child" indicator inherits it's module from its parent:

[4]:
```
tg_mean_c.__module__ == xc.atmos.tg_mean.__module__
```

[4]:
```
True
```

To create indicators with a different module, for example, in a goal to differentiate them in the registry, two methods can be used : passing `module` to the constructor, or using conventional class inheritance.

[5]:
```
# Passing module
tg_mean_c2 = registry["TG_MEAN_C"](module="test")  # we didn't change the identifier!
print(tg_mean_c2.__module__)
"test.TG_MEAN_C" in registry
```

```
xclim.indicators.test
```

[5]:
```
True
```

[6]:
```
# Conventionnal class inheritance, uses the current module name
class TG_MEAN_C3(registry["TG_MEAN_C"]):
    pass  # nothing to change really


tg_mean_c3 = TG_MEAN_C3()

print(tg_mean_c3.__module__)
"__main__.TG_MEAN_C" in registry
```

```
__main__
```

[6]:
```
True
```

While the former method is shorter, the latter is what xclim uses internally as it provides some clean code structure. See the code in the github repo.

3.6.4 Virtual modules

`xclim` gives users the ability to generate their own modules from existing indices library. These mappings can help in emulating existing libraries (such as ICCLIM), with the added benefit of CF-compliant metadata, multilingual metadata support, and optimized calculations using federated resources (using Dask). This can be used for example to tailor existing indices with predefined thresholds without having to rewrite indices.

Presently, xclim is capable of approximating the indices developed in ICCLIM, ANUCLIM and clix-meta and is open to contributions of new indices and library mappings.

This notebook serves as an example of how one might go about creating their own library of mapped indices. Two ways are possible:

1. From a YAML file (recommended way)

2. From a mapping (dictionary) of indicators

YAML file

The first method is based on the YAML syntax proposed by `clix-meta`, expanded to xclim's needs. The full documentation on that syntax is *here*. This notebook shows an example different complexities of indicator creation. It creates a minimal python module defining a indice, creates a YAML file with the metadata for several indicators and then parses it into xclim.

```
[8]: # These variables were generated by a hidden cell above that syntax-colored them.
print("Content of example.py :")
print(highlighted_py)
print("\n\nContent of example.yml :")
print(highlighted_yaml)
print("\n\nContent of example.fr.json :")
print(highlighted_json)
```

```
Content of example.py :
# noqa: D100
from __future__ import annotations

import xarray as xr

from xclim.core.units import declare_units, rate2amount


@declare_units(pr="[precipitation]")
def extreme_precip_accumulation_and_days(
    pr: xr.DataArray, perc: float = 95, freq: str = "YS"
):
    """Total precipitation accumulation during extreme events and number of days of such␣
↪precipitation.

    The `perc` percentile of the precipitation (including all values, not in a day-of-
↪year manner)
    is computed. Then, for each period, the days where `pr` is above the threshold are␣
↪accumulated,
    to get the total precip related to those extreme events.

    Parameters
    ----------
    pr: xr.DataArray
      Precipitation flux (both phases).
    perc: float
      Percentile corresponding to "extreme" precipitation, [0-100].
    freq: str
      Resampling frequency.

    Returns
    -------
    xarray.DataArray
      Precipitation accumulated during events where pr was above the {perc}th percentile␣
↪of the whole series.
```

(continues on next page)

(continued from previous page)

```python
    xarray.DataArray
      Number of days where pr was above the {perc}th percentile of the whole series.
    """
    pr_thresh = pr.quantile(perc / 100, dim="time").drop_vars("quantile")

    extreme_days = pr >= pr_thresh
    pr_extreme = rate2amount(pr).where(extreme_days)

    out1 = pr_extreme.resample(time=freq).sum()
    out1.attrs["units"] = pr_extreme.units

    out2 = extreme_days.resample(time=freq).sum()
    out2.attrs["units"] = "days"
    return out1, out2
```

Content of example.yml :

```yaml
doc: |
  ==============
  Example module
  ==============

  This module is an example of YAML generated xclim submodule.
realm: atmos
references: xclim documentation https://xclim.readthedocs.io
indicators:
  RX1day:
    base: rx1day
    cf_attrs:
      long_name: Highest 1-day precipitation amount
  RX5day:
    base: max_n_day_precipitation_amount
    cf_attrs:
      long_name: Highest 5-day precipitation amount
    parameters:
      freq: QS-DEC
      window: 5
  R75pdays:
    base: days_over_precip_thresh
    parameters:
      pr_per:
        description: Daily 75th percentile of wet day precipitation flux.
      thresh: 1 mm/day
  fd:
    compute: count_occurrences
    input:
      data: tasmin
    cf_attrs:
      cell_methods: 'time: minimum within days time: sum over days'
      long_name: Number of Frost Days (Tmin < 0°C)
      standard_name: number_of_days_with_air_temperature_below_threshold
```

(continues on next page)

(continued from previous page)

```
      units: days
      var_name: fd
    parameters:
      condition: <
      threshold: 0 degC
      freq:
        default: YS
    references: ETCCDI
  R95p:
    compute: extreme_precip_accumulation_and_days
    cf_attrs:
      - cell_methods: 'time: sum within days time: sum over days'
        long_name: Annual total PRCP when RR > {perc}th percentile
        units: m
        var_name: R95p
      - long_name: Annual number of days when RR > {perc}th percentile
        units: days
        var_name: R95p_days
    parameters:
      perc: 95
    references: climdex
  R99p:
    base: .R95p
    cf_attrs:
      - var_name: R99p
      - var_name: R99p_days
    parameters:
      perc: 99
```

```
Content of example.fr.json :
{
  "FD": {
    "title": "Nombre de jours de gel",
    "long_name": "Nombre de jours de gel (Tmin < 0°C)",
    "description": "Nombre de jours où la température minimale passe sous 0°C."
  },
  "R95P": {
    "title": "Précpitations accumulées lors des jours de fortes pluies (> {perc}e␣
↪percentile)"
  },
  "R95P.R95p": {
    "long_name": "Accumulation {freq:f} des précipitations lors des jours de fortes␣
↪pluies (> {perc}e percentile)",
    "description": "Épaisseur équivalente des précipitations accumulées lors des jours␣
↪où la pluie est plus forte que le {perc}e percentile de la série."
  },
  "R95P.R95p_days": {
    "long_name": "Nombre de jours de fortes pluies (> {perc}e percentile)",
    "description": "Nombre de jours où la pluie est plus forte que le {perc}e percentile␣
↪de la série."
```

(continues on next page)

(continued from previous page)

```
  },
  "R99P.R99p": {
    "long_name": "Accumulation {freq:f} des précipitations lors des jours de fortes␣
↪pluies (> {perc}e percentile)",
    "description": "Épaisseur équivalente des précipitations accumulées lors des jours␣
↪où la pluie est plus forte que le {perc}e percentile de la série."
  },
  "R99P.R99p_days": {
    "long_name": "Nombre de jours de fortes pluies (> {perc}e percentile)",
    "description": "Nombre de jours où la pluie est plus forte que le {perc}e percentile␣
↪de la série."
  },
  "RX5DAY": {
    "long_name": "Cumul maximal de la précipitation quotidienne sur 5 jours."
  }
}
```

example.yml created a module of 4 indicators.

Values of the base arguments are the **identifier** of the associated indicators, and those can be different than their name within the python modules. For example, xc.atmos.relative_humidity has HURS as identifier. One can always access xc.atmos.relative_humidity.identifier to get the correct name to use.

- RX1day is simply the same as registry['RX1DAY'], but with an updated long_name.
- RX5day is based on registry['MAX_N_DAY_PRECIPITATION_AMOUNT'], changed the long_name and injects the window and freq arguments.
- R75pdays is based on registry['DAYS_OVER_PRECIP_THRESH'], injects the thresh argument and changes the description of the per argument.
- fd is a more complex example. As there were no base: entry, the Daily class serves as a base. As it is pretty much empty, a lot has to be given explicitly:
  - Many output metadata fields are given
  - A compute function name if given (here it refers to a function in xclim.indices.generic).
  - Some parameters are injected, the default for freq is modified.
  - The input variable data is mapped to a known variable. Functions in xclim.indices.generic are indeed generic. Here we tell xclim that the data argument is minimum daily temperature. This will set the proper units check, default value and CF-compliance checks.
- R95p is similar to fd but here the compute is not defined in xclim but rather in example.py. Also, the custom function returns two outputs, so the output section is a list of mappings rather than a mapping directly.
- R99p is the same as R95p but changes the injected value. In order to avoid rewriting the output metadata, and allowed periods, we based it on R95p : as the latter was defined within the current yaml file, the identifier is prefixed by a dot (.).

Additionnaly, the yaml specified a realm and references to be used on all indices and provided a submodule docstring. Creating the module is then simply:

Finally, french translations for the main attributes and the new indicaters are given in `example.fr.json`. Even though new indicator objects are created for each yaml entry, non-specified translations are taken from the base classes if missing in the `json` file.

Note that all files are named the same way : `example.<ext>`, with the translations having an additionnal suffix giving the locale name. In the next cell, we build the module by passing only the path without extension. This absence of extension is what tells xclim to try to parse a module (`*.py`) and custom translations (`*.<locale>.json`). Those two could also be read beforehand and passed through the `indices=` and `translations=` arguments.

Validation of the YAML file

Using yamale, it is possible to check if the yaml file is valid. xclim ships with a schema (in `xclim/data/schema.yml`) file. The file can be located with:

[9]:
```
from importlib.resources import path

with path("xclim.data", "schema.yml") as f:
    print(f)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/data/schema.yml
```

And the validation can be executed either in a python session:

[10]:
```
import yamale

with path("xclim.data", "schema.yml") as f:
    schema = yamale.make_schema(f)
    data = yamale.make_data("example.yml")  # in the current folder
yamale.validate(schema, data)
```

[10]:
```
[<yamale.schema.validationresults.ValidationResult at 0x7f18b9374a90>]
```

No errors means it passed. The validation can also be run through the command line with:

```
yamale -s path/to/schema.yml path/to/module.yml
```

Loading the module and computating of the indices.

[11]:
```
import xclim as xc

example = xc.core.indicator.build_indicator_module_from_yaml("example", mode="raise")
```

[12]:
```
print(example.__doc__)
print("--")
print(xc.indicators.example.R99p.__doc__)
```

```
==============
Example module
==============

This module is an example of YAML generated xclim submodule.
```

(continues on next page)

(continued from previous page)

```
--
Total precipitation accumulation during extreme events and number of days of such␣
↪precipitation. (realm: atmos)

The `perc` percentile of the precipitation (including all values, not in a day-of-year␣
↪manner) is computed. Then, for each period, the days where `pr` is above the threshold␣
↪are accumulated, to get the total precip related to those extreme events.

This indicator will check for missing values according to the method "from_context".
Based on indice :py:func:`~example.extreme_precip_accumulation_and_days`.
With injected parameters: perc=99.

Parameters
----------
pr : str or DataArray
  Precipitation flux (both phases).
  Default : `ds.pr`. [Required units : [precipitation]]
freq : offset alias (string)
  Resampling frequency.
  Default : YS.
ds : Dataset, optional
  A dataset with the variables given by name.
  Default : None.

Returns
-------
R99p : DataArray
  Annual total PRCP when RR > {perc}th percentile [m]
  cell_methods: time: sum within days time: sum over days
R99p_days : DataArray
  Annual number of days when RR > {perc}th percentile [days]

References
----------
xclim documentation https://xclim.readthedocs.io
```

Useful for using this technique in large projects, we can iterate over the indicators like so:

[13]:
```
from xclim.testing import open_dataset

ds = open_dataset("ERA5/daily_surface_cancities_1990-1993.nc")
ds2 = ds.assign(
    pr_per=xc.core.calendar.percentile_doy(ds.pr, window=5, per=75).isel(
        percentiles=0, drop=True
    )
)

outs = []
with xc.set_options(metadata_locales="fr"):
```

(continues on next page)

(continued from previous page)

```
    for name, ind in example.iter_indicators():
        print(f"Indicator: {name}")
        print(f"\tIdentifier: {ind.identifier}")
        print(f"\tTitle: {ind.title}")
        out = ind(ds=ds2)  # Use all default arguments and variables from the dataset
        if isinstance(out, tuple):
            outs.extend(out)
        else:
            outs.append(out)
```

```
Indicator: RX1day
        Identifier: RX1day
        Title: Highest 1-day precipitation amount for a period (frequency).
Indicator: RX5day
        Identifier: RX5day
        Title: Highest precipitation amount cumulated over a n-day moving window.
Indicator: R75pdays
        Identifier: R75pdays
        Title: Number of wet days with daily precipitation over a given percentile.
Indicator: fd
        Identifier: fd
        Title: Calculate the number of times some condition is met.
Indicator: R95p
        Identifier: R95p
        Title: Total precipitation accumulation during extreme events and number of days␣
↪of such precipitation.
Indicator: R99p
        Identifier: R99p
        Title: Total precipitation accumulation during extreme events and number of days␣
↪of such precipitation.
```

out contains all the computed indices, with translated metadata. Note that this merge doesn't make much sense with the current list of indicators since they have different frequencies (`freq`).

[14]:
```
out = xr.merge(outs)
out.attrs = {
    "title": "Indicators computed from the example module."
}  # Merge puts the attributes of the first variable, we don't want that.
out
```

[14]:
```
<xarray.Dataset>
Dimensions:                  (time: 21, location: 5)
Coordinates:
  * time                     (time) datetime64[ns] 1989-12-01 ... 1993-12-01
    lat                      (location) float32 44.5 45.5 63.75 52.0 48.5
  * location                 (location) object 'Halifax' ... 'Victoria'
    lon                      (location) float32 -63.5 -73.5 -68.5 -106.8 -123.2
Data variables:
    RX1day                   (location, time) float32 nan 61.13 nan ... nan nan
    RX5day                   (location, time) float64 nan nan 84.1 ... 26.55 nan
    days_over_precip_thresh  (location, time) float64 nan 93.0 nan ... nan nan
    fd                       (location, time) float64 nan 92.0 nan ... nan nan
    R95p                     (location, time) float64 nan 0.7553 nan ... nan nan
```

(continues on next page)

(continued from previous page)

```
    R95p_days                  (location, time) float64 nan 24.0 nan ... nan nan
    R99p                       (location, time) float64 nan 0.2054 nan ... nan nan
    R99p_days                  (location, time) float64 nan 4.0 nan ... nan nan
Attributes:
    title:    Indicators computed from the example module.
```

Mapping of indicators

For more complex mappings, submodules can be constructed from Indicators directly. This is not the recommended way, but can sometimes be a workaround when the YAML version is lacking features.

[15]:
```
from xclim.core.indicator import build_indicator_module, registry
from xclim.core.utils import wrapped_partial

mapping = dict(
    egg_cooking_season=registry["MAXIMUM_CONSECUTIVE_WARM_DAYS"](
        module="awesome",
        compute=xc.indices.maximum_consecutive_tx_days,
        parameters=dict(thresh="35 degC"),
        long_name="Season for outdoor egg cooking.",
    ),
    fish_feeling_days=registry["WETDAYS"](
        module="awesome",
        compute=xc.indices.wetdays,
        parameters=dict(thresh="14.0 mm/day"),
        long_name="Days where we feel we are fishes",
    ),
    sweater_weather=xc.atmos.tg_min.__class__(module="awesome"),
)

awesome = build_indicator_module(
    name="awesome",
    objs=mapping,
    doc="""
        =========================
        My Awesome Custom indices
        =========================
        There are only 3 indices that really matter when you come down to brass tacks.
        This mapping library exposes them to users who want to perform real deal
        climate science.
        """,
)
```

[16]:
```
print(xc.indicators.awesome.__doc__)
```

```

        =========================
        My Awesome Custom indices
        =========================
        There are only 3 indices that really matter when you come down to brass tacks.
        This mapping library exposes them to users who want to perform real deal
```

(continues on next page)

(continued from previous page)

```
        climate science.
```

[17]:
```
# Let's look at our new awesome module
print(awesome.__doc__)
for name, ind in awesome.iter_indicators():
    print(f"{name} : {ind}")
```

```

        ==========================
        My Awesome Custom indices
        ==========================
        There are only 3 indices that really matter when you come down to brass tacks.
        This mapping library exposes them to users who want to perform real deal
        climate science.

egg_cooking_season : <xclim.indicators.awesome.MAXIMUM_CONSECUTIVE_WARM_DAYS object at␣
↪0x7f18b8a8afd0>
fish_feeling_days : <xclim.indicators.awesome.WETDAYS object at 0x7f18b9680100>
sweater_weather : <xclim.indicators.awesome.TG_MIN object at 0x7f18b9680160>
```

3.7 Statistical Downscaling and Bias-Adjustment

`xclim` provides tools and utilities to ease the bias-adjustement process through its `xclim.sdba` module. Almost all adjustment algorithms conform to the `train` - `adjust` scheme, formalized within `TrainAdjust` classes. Given a reference time series (ref), historical simulations (hist) and simulations to be adjusted (sim), any bias-adjustment method would be applied by first estimating the adjustment factors between the historical simulation and the observations series, and then applying these factors to `sim`, which could be a future simulation.

This presents examples, while a bit more info and the API are given on *this page*.

A very simple "Quantile Mapping" approach is available through the "Empirical Quantile Mapping" object. The object is created through the `.train` method of the class, and the simulation is adjusted with `.adjust`.

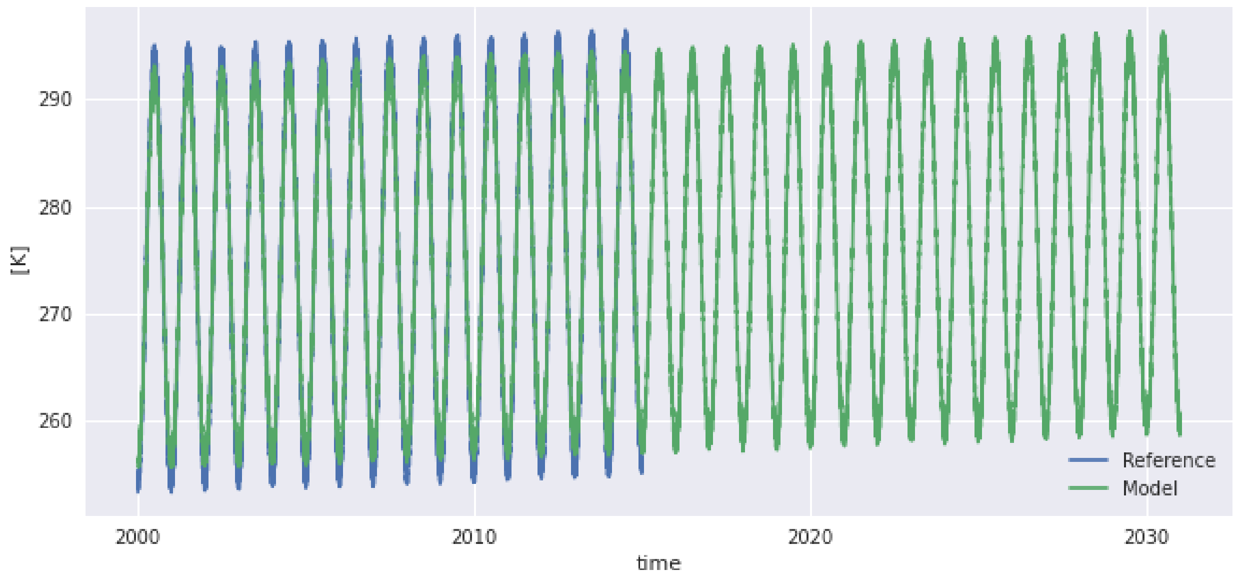
[1]:
```
from __future__ import annotations

import cftime
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr

%matplotlib inline
plt.style.use("seaborn")
plt.rcParams["figure.figsize"] = (11, 5)

# Create toy data to explore bias adjustment, here fake temperature timeseries
t = xr.cftime_range("2000-01-01", "2030-12-31", freq="D", calendar="noleap")
ref = xr.DataArray(
    (
        -20 * np.cos(2 * np.pi * t.dayofyear / 365)
```

(continues on next page)

(continued from previous page)

```
        + 2 * np.random.random_sample((t.size,))
        + 273.15
        + 0.1 * (t - t[0]).days / 365
    ),  # "warming" of 1K per decade,
    dims=("time",),
    coords={"time": t},
    attrs={"units": "K"},
)
sim = xr.DataArray(
    (
        -18 * np.cos(2 * np.pi * t.dayofyear / 365)
        + 2 * np.random.random_sample((t.size,))
        + 273.15
        + 0.11 * (t - t[0]).days / 365
    ),  # "warming" of 1.1K per decade
    dims=("time",),
    coords={"time": t},
    attrs={"units": "K"},
)

ref = ref.sel(time=slice(None, "2015-01-01"))
hist = sim.sel(time=slice(None, "2015-01-01"))

ref.plot(label="Reference")
sim.plot(label="Model")
plt.legend()
```

[1]:
```
<matplotlib.legend.Legend at 0x7f52ca29f430>
```



[2]:
```
from xclim import sdba

QM = sdba.EmpiricalQuantileMapping.train(
    ref, hist, nquantiles=15, group="time", kind="+"
```

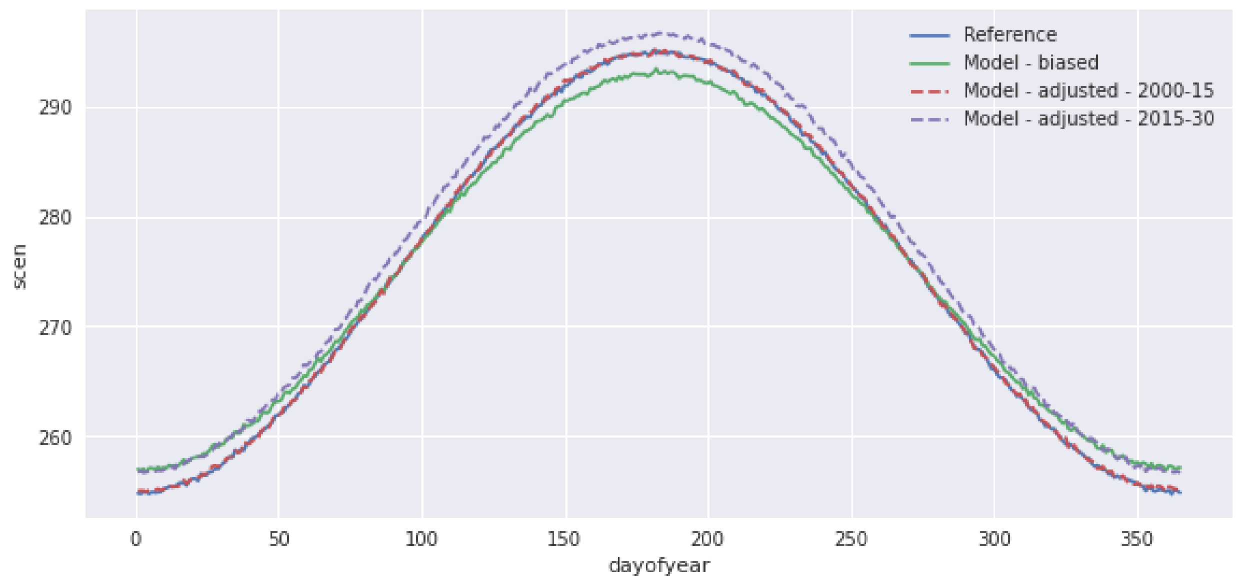
(continues on next page)

(continued from previous page)

```
)
scen = QM.adjust(sim, extrapolation="constant", interp="nearest")

ref.groupby("time.dayofyear").mean().plot(label="Reference")
hist.groupby("time.dayofyear").mean().plot(label="Model - biased")
scen.sel(time=slice("2000", "2015")).groupby("time.dayofyear").mean().plot(
    label="Model - adjusted - 2000-15", linestyle="--"
)
scen.sel(time=slice("2015", "2030")).groupby("time.dayofyear").mean().plot(
    label="Model - adjusted - 2015-30", linestyle="--"
)
plt.legend()
```

[2]:

```
<matplotlib.legend.Legend at 0x7f52c7d9c8b0>
```



In the previous example, a simple Quantile Mapping algorithm was used with 15 quantiles and one group of values. The model performs well, but our toy data is also quite smooth and well-behaved so this is not surprising. A more complex example could have biais distribution varying strongly across months. To perform the adjustment with different factors for each months, one can pass `group='time.month'`. Moreover, to reduce the risk of sharp change in the adjustment at the interface of the months, `interp='linear'` can be passed to `adjust` and the adjustment factors will be interpolated linearly. Ex: the factors for the 1st of May will be the average of those for april and those for may.

[3]:

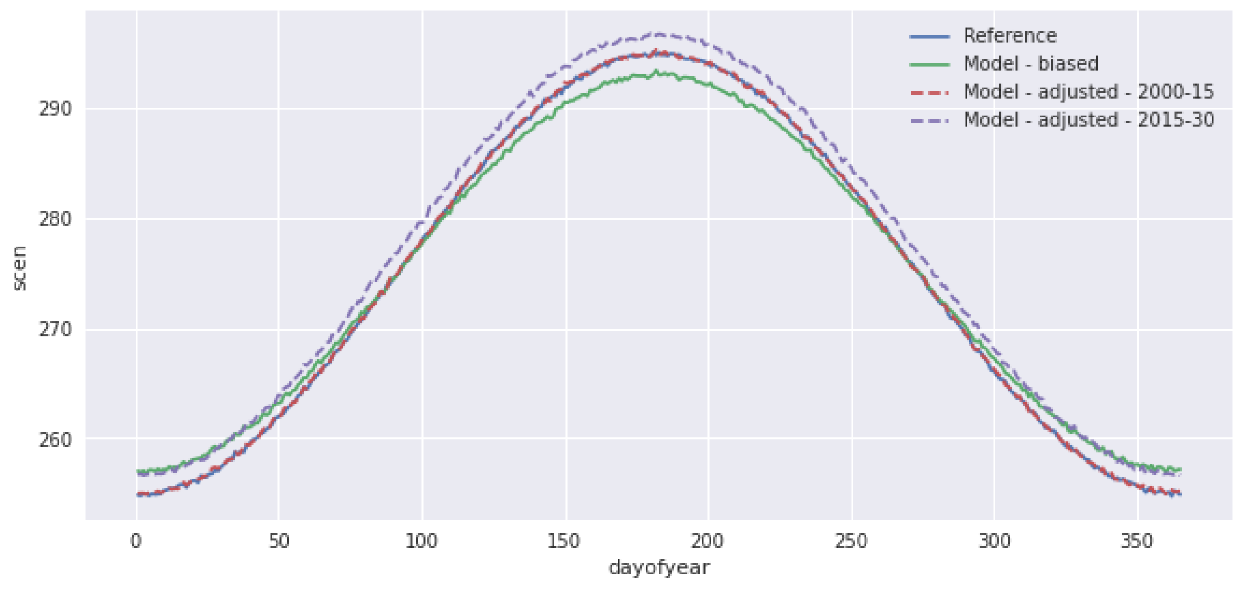
```
QM_mo = sdba.EmpiricalQuantileMapping.train(
    ref, hist, nquantiles=15, group="time.month", kind="+"
)
scen = QM_mo.adjust(sim, extrapolation="constant", interp="linear")

ref.groupby("time.dayofyear").mean().plot(label="Reference")
hist.groupby("time.dayofyear").mean().plot(label="Model - biased")
scen.sel(time=slice("2000", "2015")).groupby("time.dayofyear").mean().plot(
    label="Model - adjusted - 2000-15", linestyle="--"
)
```

(continues on next page)

(continued from previous page)

```
scen.sel(time=slice("2015", "2030")).groupby("time.dayofyear").mean().plot(
    label="Model - adjusted - 2015-30", linestyle="--"
)
plt.legend()
```

[3]:
```
<matplotlib.legend.Legend at 0x7f52c7d9c7f0>
```



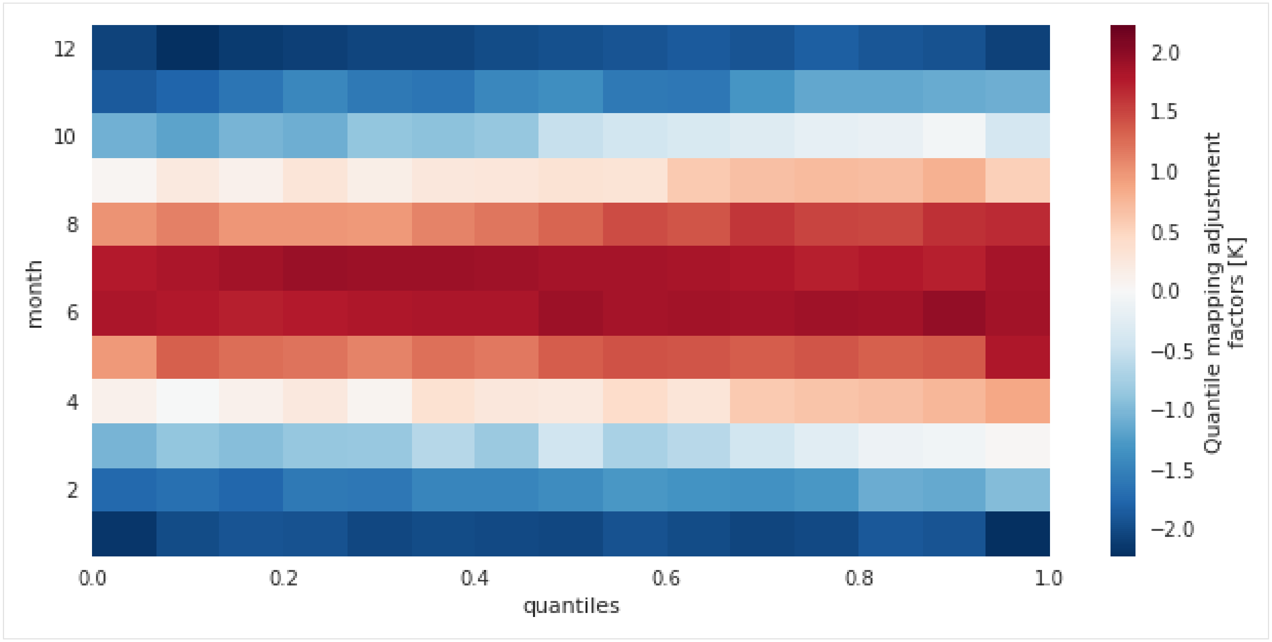
The training data (here the adjustment factors) is available for inspection in the ds attribute of the adjustment object.

[4]:
```
QM_mo.ds
```

[4]:
```
<xarray.Dataset>
Dimensions:    (quantiles: 15, month: 12)
Coordinates:
  * quantiles  (quantiles) float64 0.03333 0.1 0.1667 ... 0.8333 0.9 0.9667
  * month      (month) int64 1 2 3 4 5 6 7 8 9 10 11 12
Data variables:
    af         (month, quantiles) float64 -2.162 -1.983 -1.926 ... -1.933 -2.071
    hist_q     (month, quantiles) float64 256.1 256.4 256.7 ... 259.2 259.9
Attributes:
    group:                  time.month
    group_compute_dims:     ['time']
    group_window:           1
    _xclim_adjustment:      {"py/object": "xclim.sdba.adjustment.EmpiricalQuanti...
    adj_params:             EmpiricalQuantileMapping(group=Grouper(add_dims=[], ...
```

[5]:
```
QM_mo.ds.af.plot()
```

[5]:
```
<matplotlib.collections.QuadMesh at 0x7f52b89dad30>
```



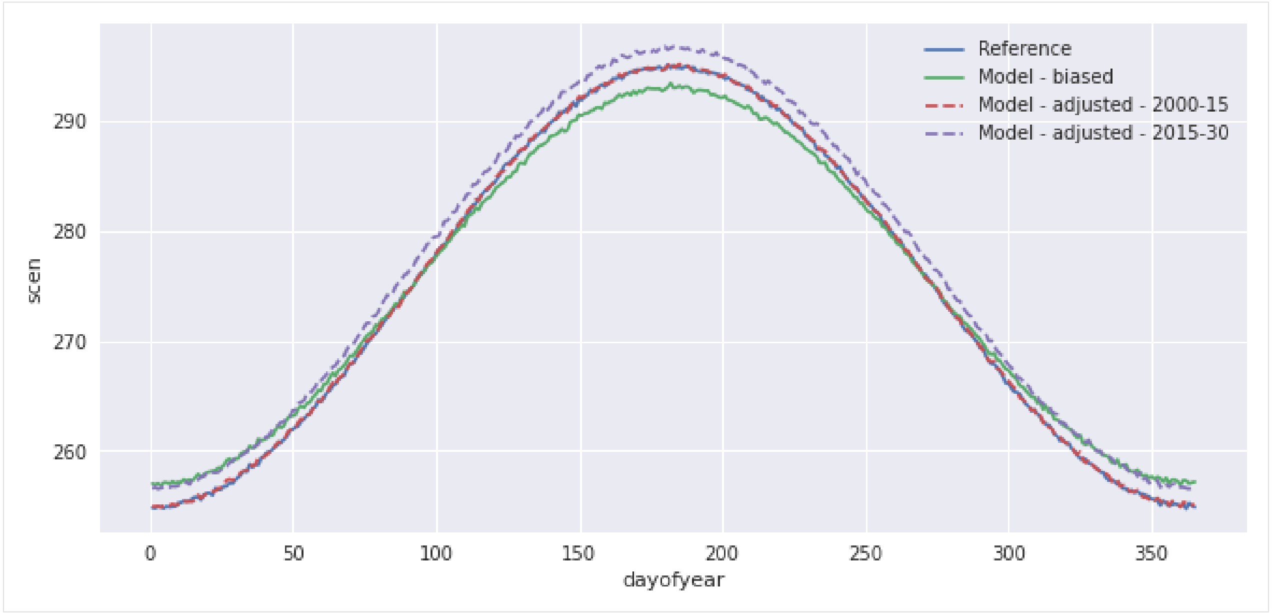
3.7.1 Grouping

For basic time period grouping (months, day of year, season), passing a string to the methods needing it is sufficient. Most methods acting on grouped data also accept a `window` int argument to pad the groups with data from adjacent ones. Units of `window` are the sampling frequency of the main grouping dimension (usually `time`). For more complex grouping, or simply for clarity, one can pass a `xclim.sdba.base.Grouper` directly.

Example here with another, simpler, adjustment method. Here we want `sim` to be scaled so that its mean fits the one of `ref`. Scaling factors are to be computed separately for each day of the year, but including 15 days on either side of the day. This means that the factor for the 1st of May is computed including all values from the 16th of April to the 15th of May (of all years).

[6]:
```
group = sdba.Grouper("time.dayofyear", window=31)
QM_doy = sdba.Scaling.train(ref, hist, group=group, kind="+")
scen = QM_doy.adjust(sim)

ref.groupby("time.dayofyear").mean().plot(label="Reference")
hist.groupby("time.dayofyear").mean().plot(label="Model - biased")
scen.sel(time=slice("2000", "2015")).groupby("time.dayofyear").mean().plot(
    label="Model - adjusted - 2000-15", linestyle="--"
)
scen.sel(time=slice("2015", "2030")).groupby("time.dayofyear").mean().plot(
    label="Model - adjusted - 2015-30", linestyle="--"
)
plt.legend()
```

[6]:
```
<matplotlib.legend.Legend at 0x7f52b8cc0d30>
```

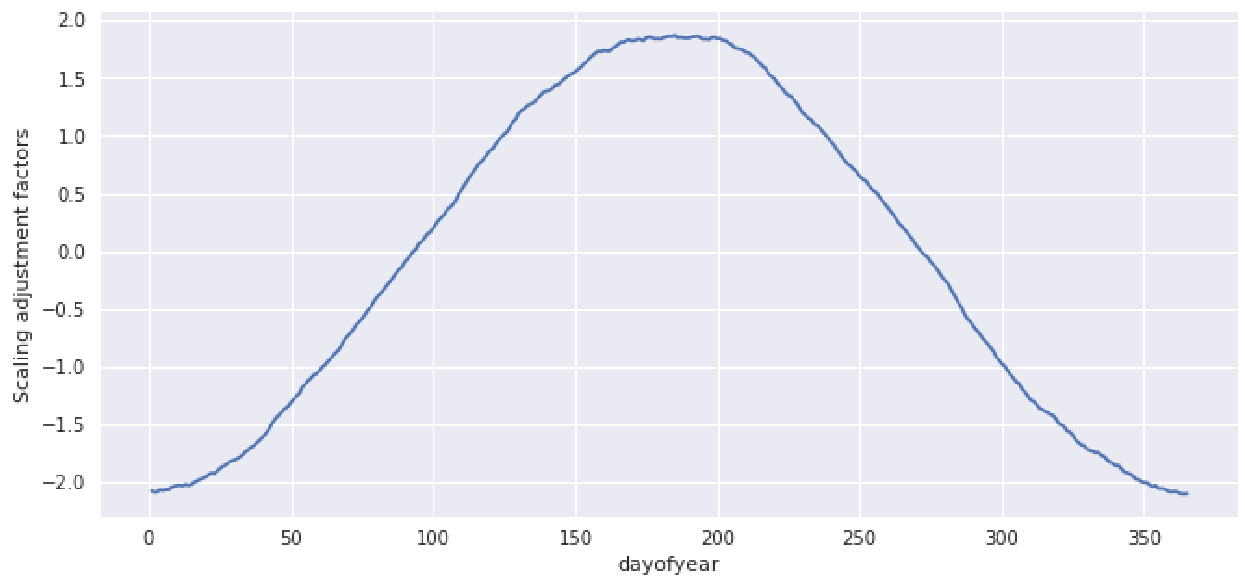


[7]:
```
sim
```

[7]:
```
<xarray.DataArray (time: 11315)>
array([256.210717, 256.469947, 256.503255, ..., 258.76469 , 259.761906,
       259.673317])
Coordinates:
  * time     (time) object 2000-01-01 00:00:00 ... 2030-12-31 00:00:00
Attributes:
    units:    K
```

[8]:
```
QM_doy.ds.af.plot()
```

[8]:
```
[<matplotlib.lines.Line2D at 0x7f52b5bbe820>]
```



3.7.2 Modular approach

The sdba module adopts a modular approach instead of implementing published and named methods directly. A generic bias adjustment process is laid out as follows:

- preprocessing on ref, hist and sim (using methods in xclim.sdba.processing or xclim.sdba.detrending)
- creating and training the adjustment object Adj = Adjustment.train(obs, hist, **kwargs) (from xclim.sdba.adjustment)
- adjustment scen = Adj.adjust(sim, **kwargs)
- post-processing on scen (for example: re-trending)

The train-adjust approach allows to inspect the trained adjustment object. The training information is stored in the underlying Adj.ds dataset and often has a af variable with the adjustment factors. Its layout and the other available variables vary between the different algorithm, refer to their part of the API docs.

For heavy processing, this separation allows the computation and writing to disk of the training dataset before performing the adjustment(s). See the *advanced notebook*.

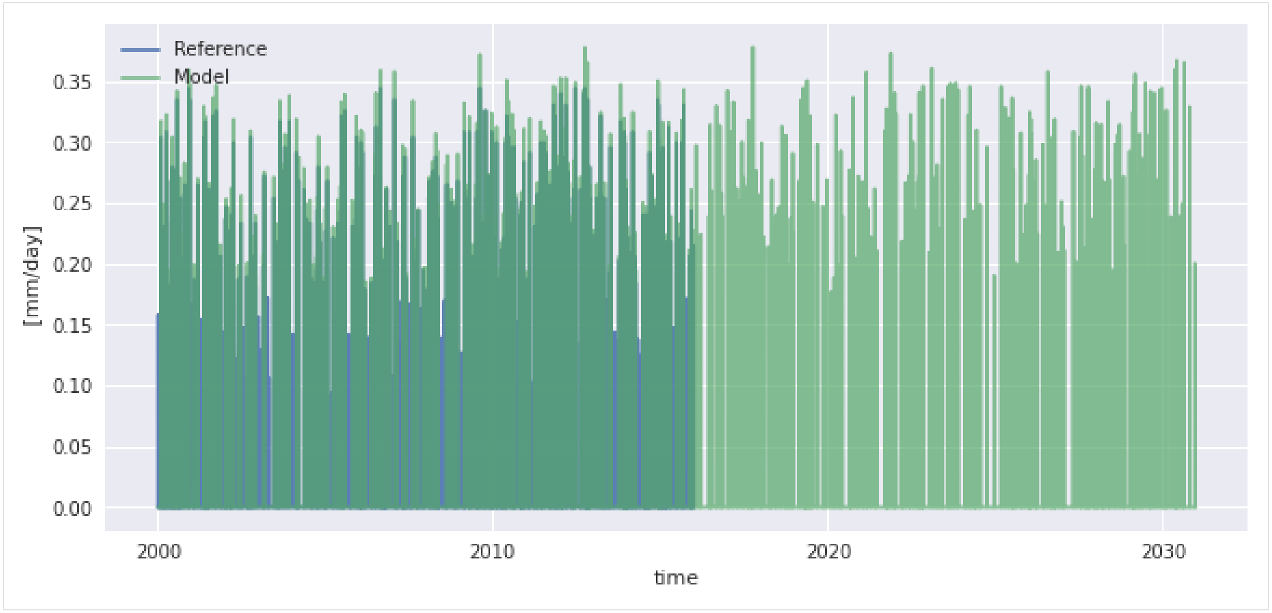
Parameters needed by the training and the adjustment are saved to the Adj.ds dataset as a adj_params attribute. Other parameters, those only needed by the adjustment are passed in the adjust call and written to the history attribute in the output scenario dataarray.

First example : pr and frequency adaptation

The next example generates fake precipitation data and adjusts the sim timeseries but also adds a step where the dry-day frequency of hist is adapted so that is fits the one of ref. This ensures well-behaved adjustment factors for the smaller quantiles. Note also that we are passing kind='*' to use the multiplicative mode. Adjustment factors will be multiplied/divided instead of being added/substracted.

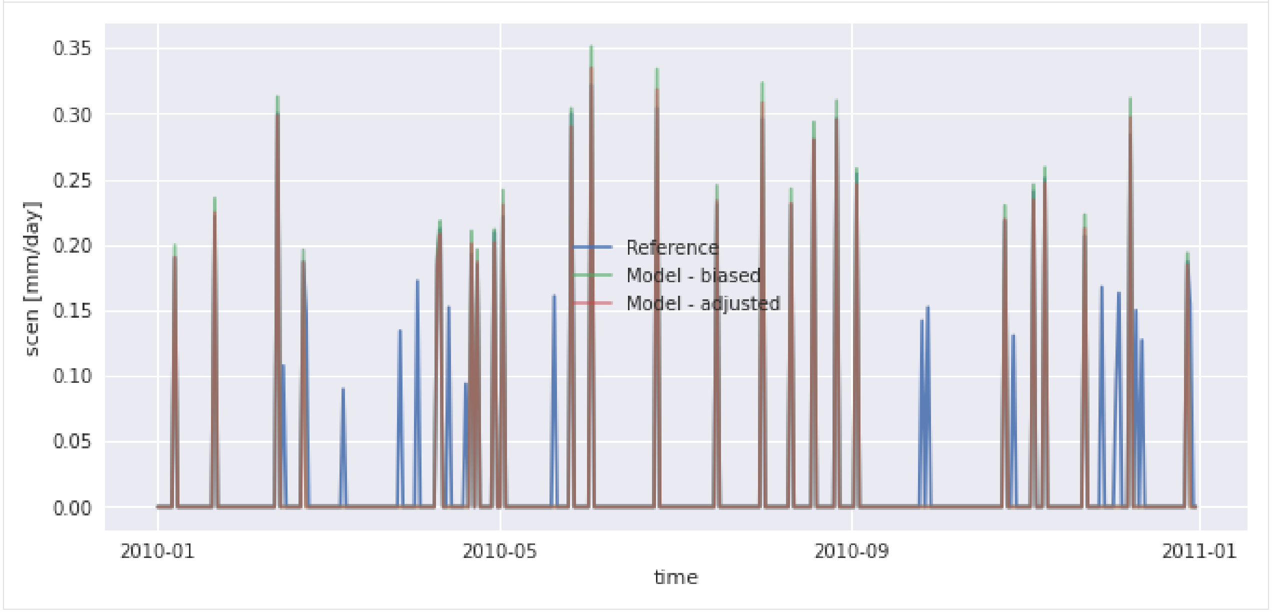
[9]:
```
vals = np.random.randint(0, 1000, size=(t.size,)) / 100
vals_ref = (4 ** np.where(vals < 9, vals / 100, vals)) / 3e6
vals_sim = (
    (1 + 0.1 * np.random.random_sample((t.size,)))
    * (4 ** np.where(vals < 9.5, vals / 100, vals))
    / 3e6
)

pr_ref = xr.DataArray(
    vals_ref, coords={"time": t}, dims=("time",), attrs={"units": "mm/day"}
)
pr_ref = pr_ref.sel(time=slice("2000", "2015"))
pr_sim = xr.DataArray(
    vals_sim, coords={"time": t}, dims=("time",), attrs={"units": "mm/day"}
)
pr_hist = pr_sim.sel(time=slice("2000", "2015"))

pr_ref.plot(alpha=0.9, label="Reference")
pr_sim.plot(alpha=0.7, label="Model")
plt.legend()
```

[9]:
```
<matplotlib.legend.Legend at 0x7f52b5bd9100>
```



[10]:
```
# 1st try without adapt_freq
QM = sdba.EmpiricalQuantileMapping.train(
    pr_ref, pr_hist, nquantiles=15, kind="*", group="time"
)
scen = QM.adjust(pr_sim)

pr_ref.sel(time="2010").plot(alpha=0.9, label="Reference")
pr_hist.sel(time="2010").plot(alpha=0.7, label="Model - biased")
scen.sel(time="2010").plot(alpha=0.6, label="Model - adjusted")
plt.legend()
```

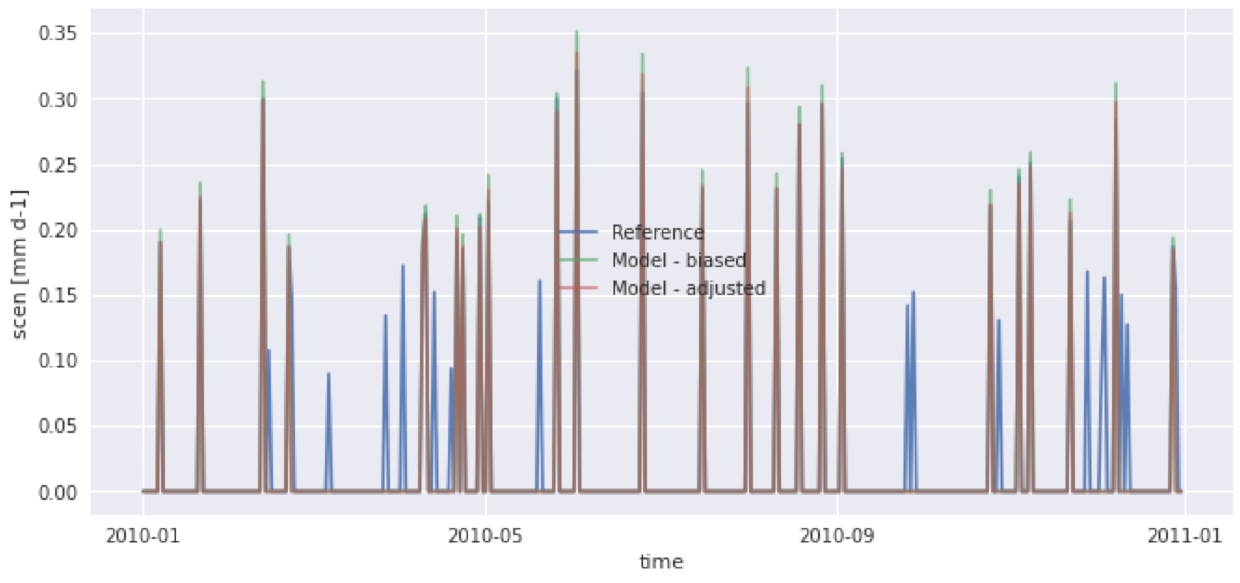
[10]:
```
<matplotlib.legend.Legend at 0x7f52b5be36d0>
```



In the figure above, scen has small peaks where sim is 0. This problem originates from the fact that there are more "dry days" (days with almost no precipitation) in hist than in ref. The next example works around the problem using frequency-adaptation, as described in Themeßl et al. (2010).

```
[11]: # 2nd try with adapt_freq
sim_ad, pth, dP0 = sdba.processing.adapt_freq(
    pr_ref, pr_sim, thresh="0.05 mm d-1", group="time"
)
QM_ad = sdba.EmpiricalQuantileMapping.train(
    pr_ref, sim_ad, nquantiles=15, kind="*", group="time"
)
scen_ad = QM_ad.adjust(pr_sim)

pr_ref.sel(time="2010").plot(alpha=0.9, label="Reference")
pr_sim.sel(time="2010").plot(alpha=0.7, label="Model - biased")
scen_ad.sel(time="2010").plot(alpha=0.6, label="Model - adjusted")
plt.legend()
```

```
[11]: <matplotlib.legend.Legend at 0x7f52b5b69250>
```



Second example: tas and detrending

The next example reuses the fake temperature timeseries generated at the beginning and applies the same QM adjustment method. However, for a better adjustment, we will scale sim to ref and then detrend the series, assuming the trend is linear. When `sim` (or `sim_scl`) is detrended, its values are now anomalies, so we need to normalize `ref` and `hist` so we can compare similar values.

This process is detailed here to show how the sdba module should be used in custom adjustment processes, but this specific method also exists as `sdba.DetrendedQuantileMapping` and is based on Cannon et al. 2015. However, `DetrendedQuantileMapping` normalizes over a `time.dayofyear` group, regardless of what is passed in the `group` argument. As done here, it is anyway recommended to use `dayofyear` groups when normalizing, especially for variables with strong seasonal variations.

```
[12]: doy_win31 = sdba.Grouper("time.dayofyear", window=15)
Sca = sdba.Scaling.train(ref, hist, group=doy_win31, kind="+")
sim_scl = Sca.adjust(sim)
```

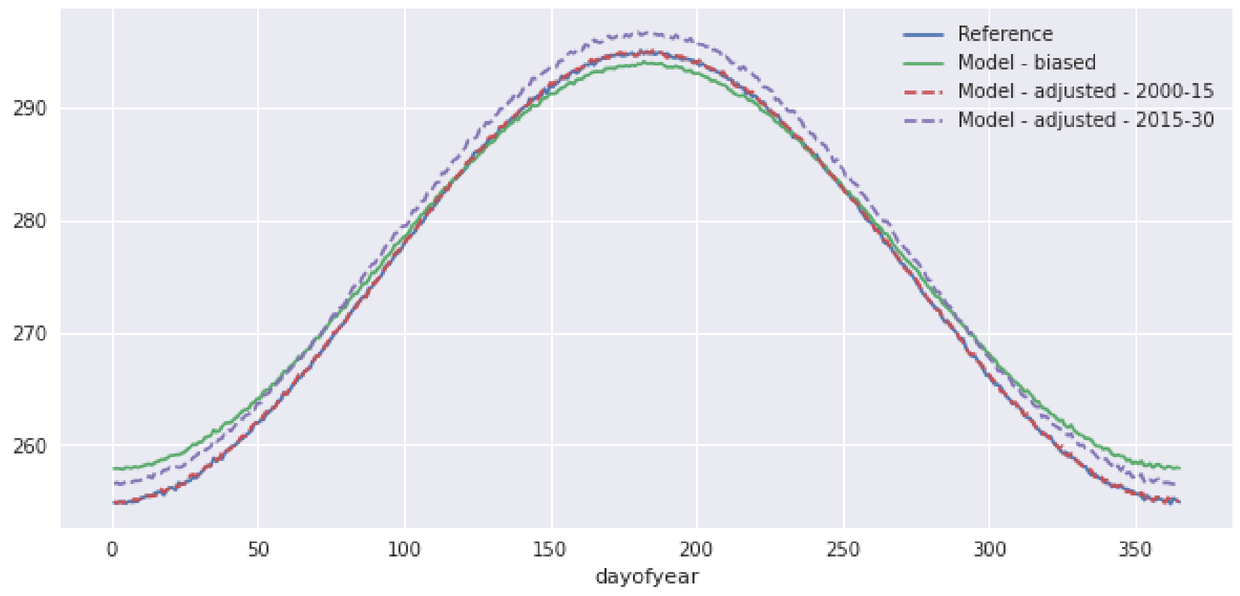
(continues on next page)

(continued from previous page)

```
detrender = sdba.detrending.PolyDetrend(degree=1, group="time.dayofyear", kind="+")
sim_fit = detrender.fit(sim_scl)
sim_detrended = sim_fit.detrend(sim_scl)

ref_n, _ = sdba.processing.normalize(ref, group=doy_win31, kind="+")
hist_n, _ = sdba.processing.normalize(hist, group=doy_win31, kind="+")

QM = sdba.EmpiricalQuantileMapping.train(
    ref_n, hist_n, nquantiles=15, group="time.month", kind="+"
)
scen_detrended = QM.adjust(sim_detrended, extrapolation="constant", interp="nearest")
scen = sim_fit.retrend(scen_detrended)


ref.groupby("time.dayofyear").mean().plot(label="Reference")
sim.groupby("time.dayofyear").mean().plot(label="Model - biased")
scen.sel(time=slice("2000", "2015")).groupby("time.dayofyear").mean().plot(
    label="Model - adjusted - 2000-15", linestyle="--"
)
scen.sel(time=slice("2015", "2030")).groupby("time.dayofyear").mean().plot(
    label="Model - adjusted - 2015-30", linestyle="--"
)
plt.legend()
```

[12]:

```
<matplotlib.legend.Legend at 0x7f52b5b38cd0>
```



Third example : Multi-method protocol - Hnilica et al. 2017

In their paper of 2017, Hnilica, Hanel and Pus present a bias-adjustment method based on the principles of Principal Components Analysis. The idea is simple : use principal components to define coordinates on the reference and on the simulation and then transform the simulation data from the latter to the former. Spatial correlation can thus be conserved by taking different points as the dimensions of the transform space. The method was demonstrated in the article by bias-adjusting precipitation over different drainage basins.

The same method could be used for multivariate adjustment. The principle would be the same, concatening the different variables into a single dataset along a new dimension. An example is given in the *advanced notebook*.

Here we show how the modularity of `xclim.sdba` can be used to construct a quite complex adjustment protocol involving two adjustment methods : quantile mapping and principal components. Evidently, as this example uses only 2 years of data, it is not complete. It is meant to show how the adjustment functions and how the API can be used.

[13]:
```
# We are using xarray's "air_temperature" dataset
ds = xr.tutorial.open_dataset("air_temperature")
```

[14]:
```
# To get an exagerated example we select different points
# here "lon" will be our dimension of two "spatially correlated" points
reft = ds.air.isel(lat=21, lon=[40, 52]).drop_vars(["lon", "lat"])
simt = ds.air.isel(lat=18, lon=[17, 35]).drop_vars(["lon", "lat"])

# Principal Components Adj, no grouping and use "lon" as the space dimensions
PCA = sdba.PrincipalComponents.train(reft, simt, group="time", crd_dim="lon")
scen1 = PCA.adjust(simt)

# QM, no grouping, 20 quantiles and additive adjustment
EQM = sdba.EmpiricalQuantileMapping.train(
    reft, scen1, group="time", nquantiles=50, kind="+"
)
scen2 = EQM.adjust(scen1)
```

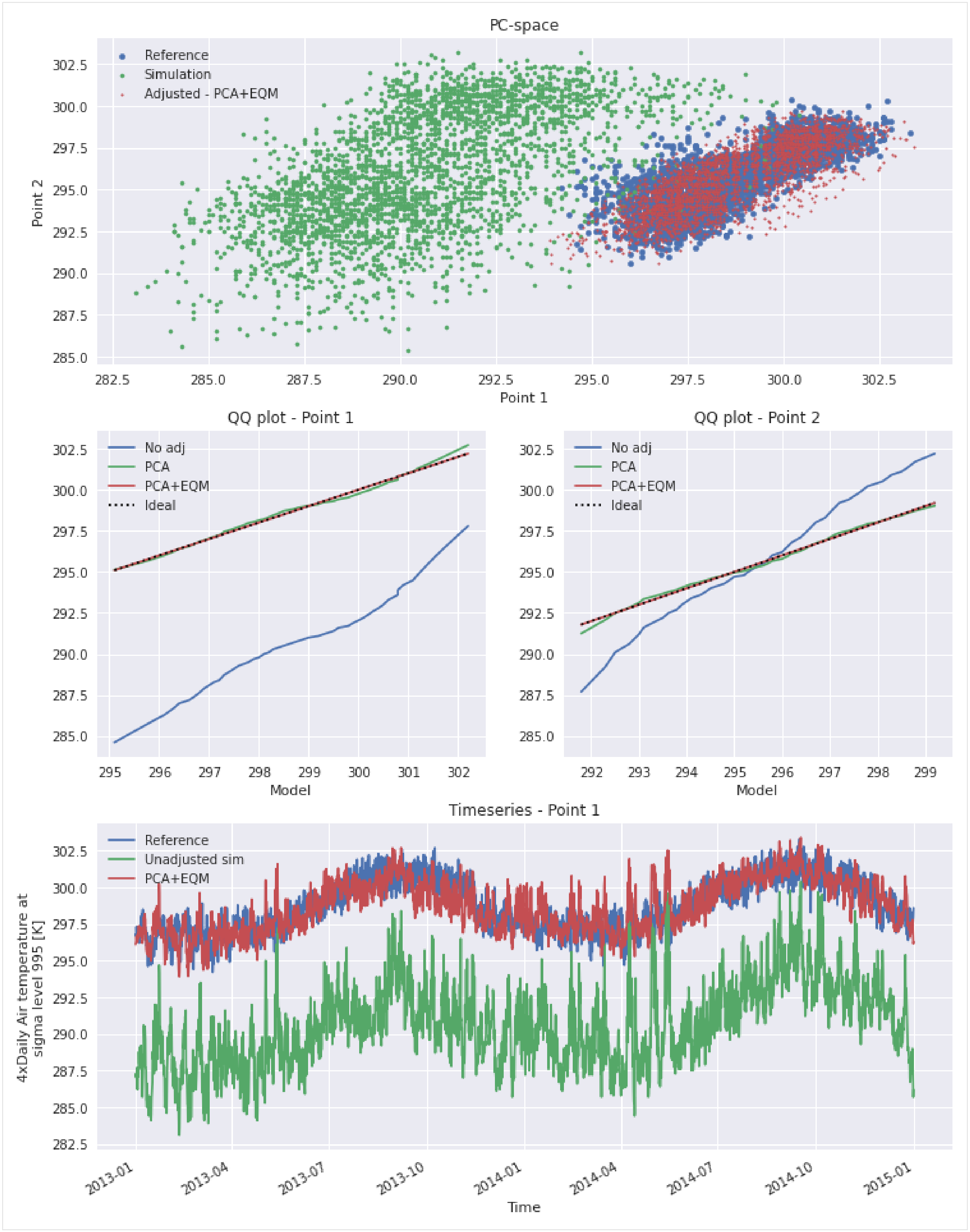
[15]:
```
# some Analysis figures
fig = plt.figure(figsize=(12, 16))
gs = plt.matplotlib.gridspec.GridSpec(3, 2, fig)

axPCA = plt.subplot(gs[0, :])
axPCA.scatter(reft.isel(lon=0), reft.isel(lon=1), s=20, label="Reference")
axPCA.scatter(simt.isel(lon=0), simt.isel(lon=1), s=10, label="Simulation")
axPCA.scatter(scen2.isel(lon=0), scen2.isel(lon=1), s=3, label="Adjusted - PCA+EQM")
axPCA.set_xlabel("Point 1")
axPCA.set_ylabel("Point 2")
axPCA.set_title("PC-space")
axPCA.legend()

refQ = reft.quantile(EQM.ds.quantiles, dim="time")
simQ = simt.quantile(EQM.ds.quantiles, dim="time")
scen1Q = scen1.quantile(EQM.ds.quantiles, dim="time")
scen2Q = scen2.quantile(EQM.ds.quantiles, dim="time")
for i in range(2):
    if i == 0:
```

(continues on next page)

(continued from previous page)

```
        axQM = plt.subplot(gs[1, 0])
    else:
        axQM = plt.subplot(gs[1, 1], sharey=axQM)
    axQM.plot(refQ.isel(lon=i), simQ.isel(lon=i), label="No adj")
    axQM.plot(refQ.isel(lon=i), scen1Q.isel(lon=i), label="PCA")
    axQM.plot(refQ.isel(lon=i), scen2Q.isel(lon=i), label="PCA+EQM")
    axQM.plot(
        refQ.isel(lon=i), refQ.isel(lon=i), color="k", linestyle=":", label="Ideal"
    )
    axQM.set_title(f"QQ plot - Point {i + 1}")
    axQM.set_xlabel("Reference")
    axQM.set_xlabel("Model")
    axQM.legend()

axT = plt.subplot(gs[2, :])
reft.isel(lon=0).plot(ax=axT, label="Reference")
simt.isel(lon=0).plot(ax=axT, label="Unadjusted sim")
# scen1.isel(lon=0).plot(ax=axT, label='PCA only')
scen2.isel(lon=0).plot(ax=axT, label="PCA+EQM")
axT.legend()
axT.set_title("Timeseries - Point 1")
```

[15]:
```
Text(0.5, 1.0, 'Timeseries - Point 1')
```


PC-space
Reference
Simulation
Adjusted - PCA+EQM
Point 1
Point 2
QQ plot - Point 1
QQ plot - Point 2
No adj
PCA
PCA+EQM
Ideal
Model
Timeseries - Point 1
Reference
Unadjusted sim
PCA+EQM
4xDaily Air temperature at sigma level 995 [K]
Time

Fourth example : Multivariate bias-adjustment with multiple steps - Cannon 2018

This section replicates the "MBCn" algorithm described by Cannon (2018). The method relies on some univariate algorithm, an adaption of the N-pdf transform of Pitié et al. (2005) and a final reordering step.

In the following, we use the AHCCD and CanESM2 data are reference and simulation and we correct both pr and tasmax together.

[16]:
```
from xclim.core.units import convert_units_to
from xclim.testing import open_dataset

dref = open_dataset(
    "sdba/ahccd_1950-2013.nc", chunks={"location": 1}, drop_variables=["lat", "lon"]
).sel(time=slice("1981", "2010"))
dref = dref.assign(
    tasmax=convert_units_to(dref.tasmax, "K"),
    pr=convert_units_to(dref.pr, "kg m-2 s-1"),
)
dsim = open_dataset(
    "sdba/CanESM2_1950-2100.nc", chunks={"location": 1}, drop_variables=["lat", "lon"]
)

dhist = dsim.sel(time=slice("1981", "2010"))
dsim = dsim.sel(time=slice("2041", "2070"))
dref
```

[16]:
```
<xarray.Dataset>
Dimensions:   (location: 3, time: 10950)
Coordinates:
  * time      (time) object 1981-01-01 00:00:00 ... 2010-12-31 00:00:00
  * location  (location) object 'Vancouver' 'Kugluktuk' 'Amos'
Data variables:
    tasmax    (location, time) float32 dask.array<chunksize=(1, 10950), meta=np.ndarray>
    pr        (location, time) float32 dask.array<chunksize=(1, 10950), meta=np.ndarray>
Attributes:
    title:         Test dataset for xclim.sdba - observed data
    description:   Extracted from homogenized observation data (AHCCD).'Vancouv...
    comment:       'Vancouver' has tasmax from station 1108380 and pr from 110...
    history:       2021-04-23T13:30:00 Extracted from AHCCD gen2 and gen3 data.
    conventions:   CF-1.8
```

Perform an initial univariate adjustment.

[17]:
```
# additive for tasmax
QDMtx = sdba.QuantileDeltaMapping.train(
    dref.tasmax, dhist.tasmax, nquantiles=20, kind="+", group="time"
)
# Adjust both hist and sim, we'll feed both to the Npdf transform.
scenh_tx = QDMtx.adjust(dhist.tasmax)
scens_tx = QDMtx.adjust(dsim.tasmax)

# remove == 0 values in pr:
```

(continues on next page)

(continued from previous page)

```
dref["pr"] = sdba.processing.jitter_under_thresh(dref.pr, "0.01 mm d-1")
dhist["pr"] = sdba.processing.jitter_under_thresh(dhist.pr, "0.01 mm d-1")
dsim["pr"] = sdba.processing.jitter_under_thresh(dsim.pr, "0.01 mm d-1")

# multiplicative for pr
QDMpr = sdba.QuantileDeltaMapping.train(
    dref.pr, dhist.pr, nquantiles=20, kind="*", group="time"
)
# Adjust both hist and sim, we'll feed both to the Npdf transform.
scenh_pr = QDMpr.adjust(dhist.pr)
scens_pr = QDMpr.adjust(dsim.pr)

scenh = xr.Dataset(dict(tasmax=scenh_tx, pr=scenh_pr))
scens = xr.Dataset(dict(tasmax=scens_tx, pr=scens_pr))
```

Stack the variables to multivariate arrays and standardize them

The standardization process ensure the mean and standard deviation of each column (variable) is 0 and 1 respectively.

`hist` and `sim` are standardized together so the two series are coherent. We keep the mean and standard deviation to be reused when we build the result.

[18]:
```
# Stack the variables (tasmax and pr)
ref = sdba.processing.stack_variables(dref)
scenh = sdba.processing.stack_variables(scenh)
scens = sdba.processing.stack_variables(scens)

# Standardize
ref, _, _ = sdba.processing.standardize(ref)

allsim, savg, sstd = sdba.processing.standardize(xr.concat((scenh, scens), "time"))
hist = allsim.sel(time=scenh.time)
sim = allsim.sel(time=scens.time)
```

Perform the N-dimensional probability density function transform

The NpdfTransform will iteratively randomly rotate our arrays in the "variables" space and apply the univariate adjustment before rotating it back. In Cannon (2018) and Pitié et al. (2005), it can be seen that the source array's joint distribution converges toward the target's joint distribution when a large number of iterations is done.

[19]:
```
from xclim import set_options

# See the advanced notebook for details on how this option work
with set_options(sdba_extra_output=True):
    out = sdba.adjustment.NpdfTransform.adjust(
        ref,
        hist,
        sim,
```

(continues on next page)

(continued from previous page)

```
        base=sdba.QuantileDeltaMapping,  # Use QDM as the univariate adjustment.
        base_kws={"nquantiles": 20, "group": "time"},
        n_iter=20,  # perform 20 iteration
        n_escore=1000,  # only send 1000 points to the escore metric (it is realy slow)
    )

scenh = out.scenh.rename(time_hist="time")  # Bias-adjusted historical period
scens = out.scen  # Bias-adjusted future period
extra = out.drop_vars(["scenh", "scen"])

# Un-standardize (add the mean and the std back)
scenh = sdba.processing.unstandardize(scenh, savg, sstd)
scens = sdba.processing.unstandardize(scens, savg, sstd)
```

Restoring the trend

The NpdfT has given us new "hist" and "sim" arrays with a correct rank structure. However, the trend is lost in this process. We reorder the result of the initial adjustment according to the rank structure of the NpdfT outputs to get our final bias-adjusted series.

`sdba.processing.reordering` : 'ref' the argument that provides the order, 'sim' is the argument to reorder.

[20]:
```
scenh = sdba.processing.reordering(hist, scenh, group="time")
scens = sdba.processing.reordering(sim, scens, group="time")
```

[21]:
```
scenh = sdba.processing.unstack_variables(scenh)
scens = sdba.processing.unstack_variables(scens)
```

There we are!

Let's trigger all the computations. Here we write the data to disk and use `compute=False` in order to trigger the whole computation tree only once. There seems to be no way in xarray to do the same with a `load` call.

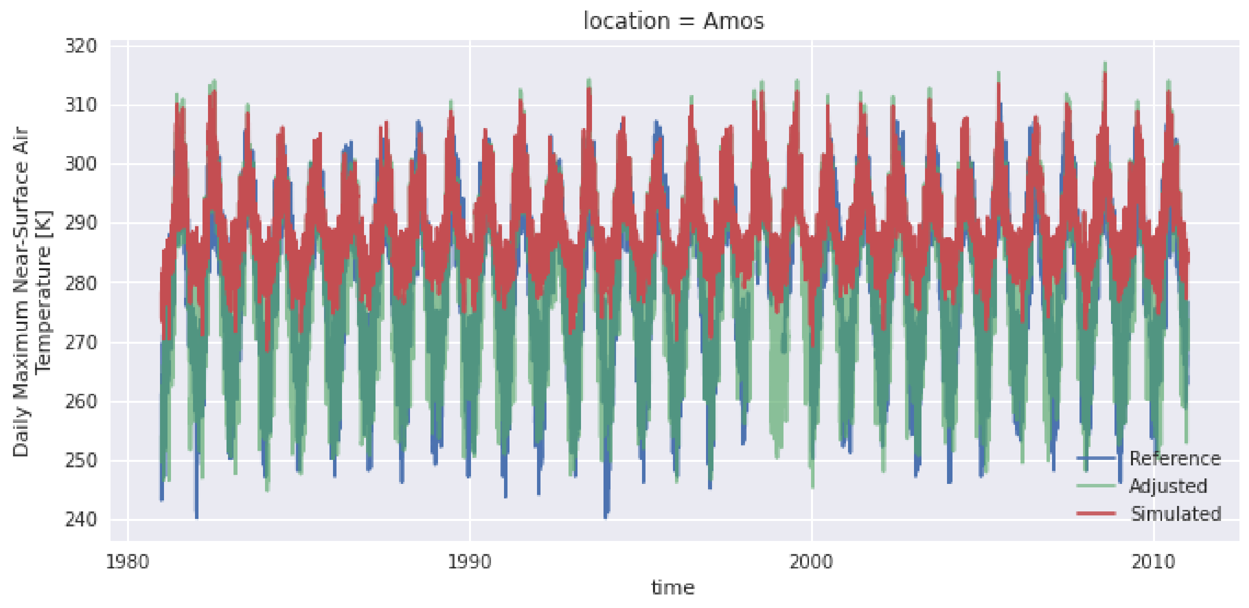
[22]:
```
from dask import compute
from dask.diagnostics import ProgressBar

tasks = [
    scenh.isel(location=2).to_netcdf("mbcn_scen_hist_loc2.nc", compute=False),
    scens.isel(location=2).to_netcdf("mbcn_scen_sim_loc2.nc", compute=False),
    extra.escores.isel(location=2)
    .to_dataset()
    .to_netcdf("mbcn_escores_loc2.nc", compute=False),
]

with ProgressBar():
    compute(tasks)
```

```
[########################################] | 100% Completed |  1min 15.3s
```

Let's compare the series and look at the distance scores to see how well the Npdf transform has converged.

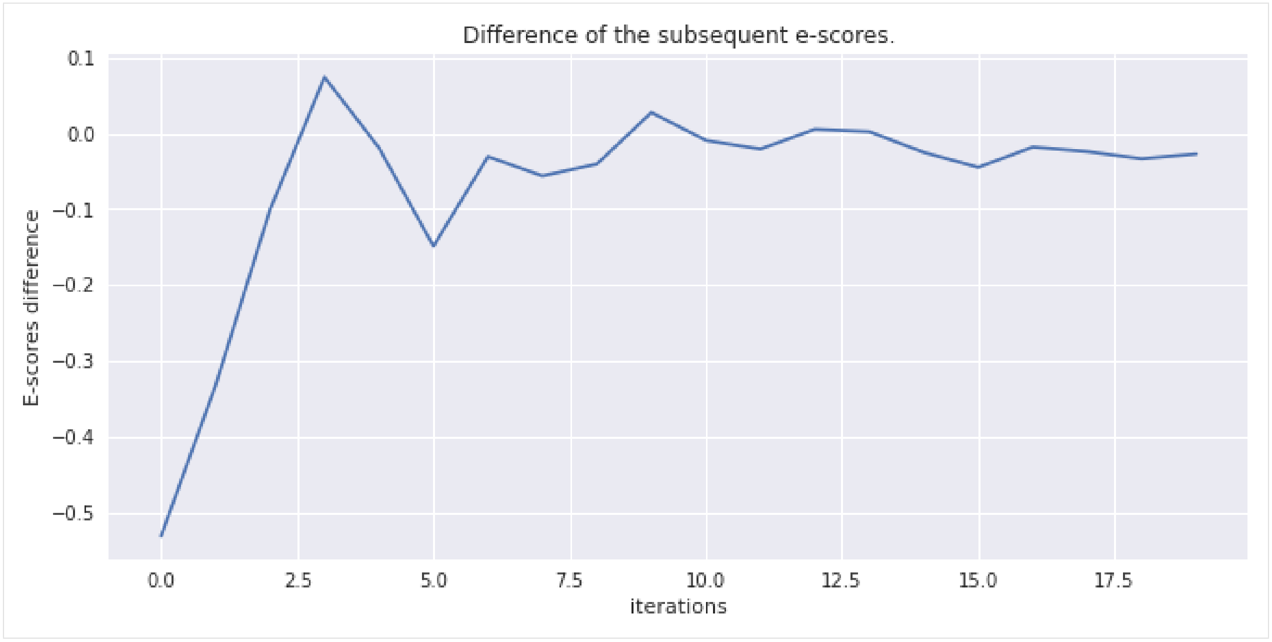
```
[23]: scenh = xr.open_dataset("mbcn_scen_hist_loc2.nc")

      fig, ax = plt.subplots()

      dref.isel(location=2).tasmax.plot(ax=ax, label="Reference")
      scenh.tasmax.plot(ax=ax, label="Adjusted", alpha=0.65)
      dhist.isel(location=2).tasmax.plot(ax=ax, label="Simulated")

      ax.legend()
```

```
[23]: <matplotlib.legend.Legend at 0x7f52b9519550>
```



```
[24]: escores = xr.open_dataarray("mbcn_escores_loc2.nc")
      diff_escore = escores.differentiate("iterations")
      diff_escore.plot()
      plt.title("Difference of the subsequent e-scores.")
      plt.ylabel("E-scores difference")
```

```
[24]: Text(0, 0.5, 'E-scores difference')
```



```
[25]: diff_escore
```

```
[25]: <xarray.DataArray 'escores' (iterations: 20)>
array([-0.5310646 , -0.3326894 , -0.10089475,  0.07398564, -0.01906785,
       -0.14894056, -0.03071404, -0.05620533, -0.04052281,  0.02741548,
       -0.00927353, -0.02080396,  0.00525162,  0.00215563, -0.0251441 ,
       -0.04481959, -0.01818538, -0.02408803, -0.03350618, -0.02719843],
      dtype=float32)
Coordinates:
  * iterations  (iterations) int64 0 1 2 3 4 5 6 7 8 ... 12 13 14 15 16 17 18 19
    location    object ...
```

The tutorial continues in the *advanced notebook* with more on optimization with dask, other fancier detrending algorithms and an example pipeline for heavy processing.

3.8 Statistical Downscaling and Bias-Adjustment - Advanced tools

The previous notebook covered the most common utilities of `xclim.sdba` for conventionnal cases. Here we explore more advanced usage of `xclim.sdba` tools.

3.8.1 Optimization with dask

Adjustment processes can be very heavy when we need to compute them over large regions and long time-series. Using small groupings (like `time.dayofyear`) adds precision and robustness, but also decuplates the load and computing complexity. Fortunately, unlike the heroic pioneers of scientific computing who managed to write parallelized Fortran, we now have dask. With only a few parameters, we can magically distribute the computing load to multiple workers and threads.

A good first read on the use of dask within xarray are the latter's Optimization tips.

Some xclim.sdba-specific tips:

- Most adjustment method will need to perform operation on the whole `time` coordinate, so it is best to optimize chunking along the other dimensions. This is often different from how public data is shared, where more universal 3D chunks are used.

  Chunking of outputs can be controlled in xarray's to_netcdf. We also suggest using Zarr files. According to its creators, `zarr` stores should give better performances, especially because of their better ability for parallel I/O. See Dataset.to_zarr and this useful rechunking package.

- One of the main bottleneck for adjustments with small groups is that dask needs to build and optimize an enormous task graph. This issue has been greatly reduced with xclim 0.27 and the use of `map_blocks` in the adjustment methods. However, not all adjustment methods use this optimized syntax.

  In order to help dask, one can split the processing in parts. For splitting traning and adjustment, see *the section below*.

- Another massive bottleneck of parallelization of xarray is the thread-locking behaviour of some methods. It is quite difficult to isolate and avoid those lockings, so one of the best workaround is to use Dask configurations with many *processes* and few *threads*. The former do not share memory and thus are not impacted when a lock is activated from a thread in another worker. However, this adds many memory transfer operations and, by experience, reduces dask's ability to parallelize some pipelines. Such a dask Client is usually created with a large `n_workers` and a small `threads_per_worker`.

- Sometimes, datasets have auxiliary coordinates (for example : lat / lon in a rotated pole dataset). Xarray handles these variables as data variables and will **not** load them if dask is used. However, in some operations, xclim or xarray will trigger an access to those variables, triggering computations each time, since they are dask-backed. To avoid this behaviour, one can load the coordinates, or simply remove them from the inputs.

3.8.2 LOESS smoothing and detrending

As described in Cleveland (1979), locally weighted linear regressions are multiple regression methods using a nearest-neighbor approach. Instead of using all data points to compute a linear or polynomial regression, LOESS algorithms compute a local regression for each point in the dataset, using only the k-nearest neighbors as selected by a weighting function. This weighting function must fulfill some strict requirements, see the doc of `xclim.sdba.loess.loess_smoothing` for more details.

In xclim's implementation, the user can choose between local *constancy* ($d = 0$, local estimates are weighted averages) and local *linearity* ($d = 1$, local estimates are taken from linear regressions). Two weighting functions are currently implemented : "tricube" ($w(x) = (1 - x^3)^3$) and "gaussian" ($w(x) = e^{-x^2/2\sigma^2}$). Finally, the number of Cleveland's *robustifying iterations* is controllable through `niter`. After computing an estimate of $y(x)$, the weights are modulated by a function of the distance between the estimate and the points and the procedure is started over. These iterations are made to weaken the effect of outliers on the estimate.

The next example shows the application of the LOESS to daily temperature data. The black line and dot are the estimated y, outputs of the `sdba.loess.loess_smoothing` function, using local linear regression (passing $d = 1$), a window spanning 20% ($f = 0.2$) of the domain, the "tricube" weighting function and only one iteration. The red curve illustrates the weighting function on January 1st 2014, where the red circles are the nearest-neighbors used in the estimation.

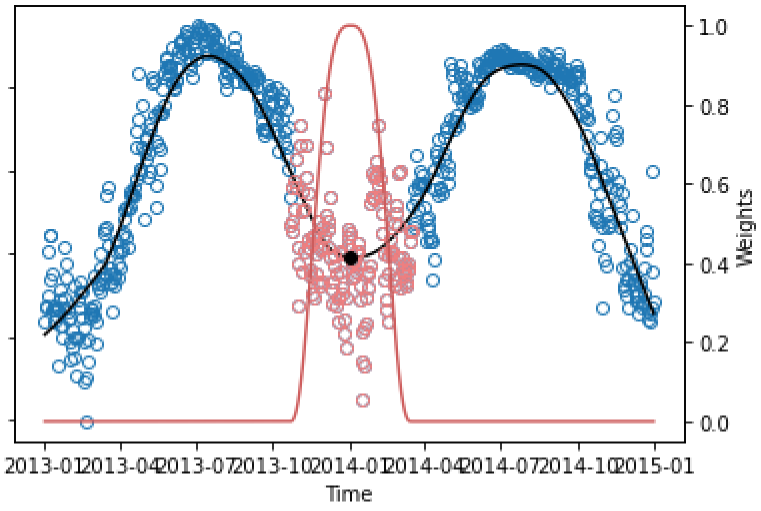
[1]:
```
from __future__ import annotations

import matplotlib.pyplot as plt
import numpy as np
import xarray as xr

```

(continues on next page)

(continued from previous page)

```
from xclim.sdba import loess

%matplotlib inline
```

[2]:
```
# Daily temperature data from xarray's tutorials
ds = xr.tutorial.open_dataset("air_temperature").resample(time="D").mean()
tas = ds.isel(lat=0, lon=0).air

# Compute the smoothed series
f = 0.2
ys = loess.loess_smoothing(tas, d=1, weights="tricube", f=f, niter=1)

# Plot data points and smoothed series
fig, ax = plt.subplots()
ax.plot(tas.time, tas, "o", fillstyle="none")
ax.plot(tas.time, ys, "k")
ax.set_xlabel("Time")
ax.set_ylabel("Temperature [K]")

## The code below calls internal functions to demonstrate how the weights are computed.

# LOESS algorithms as implemented here use scaled coordinates.
x = tas.time
x = (x - x[0]) / (x[-1] - x[0])
xi = x[366]
ti = tas.time[366]

# Weighting function take the distance with all neighbors scaled by the r parameter as␣
↪input
r = int(f * tas.time.size)
h = np.sort(np.abs(x - xi))[r]
weights = loess._tricube_weighting(np.abs(x - xi).values / h)

# Plot nearest neighbors and weighing function
wax = ax.twinx()
wax.plot(tas.time, weights, color="indianred")
ax.plot(
    tas.time, tas.where(tas * weights > 0), "o", color="lightcoral", fillstyle="none"
)

ax.plot(ti, ys[366], "ko")
wax.set_ylabel("Weights")
plt.show()
```



LOESS smoothing can suffer from heavy boundary effects. On the previous graph, we can associate the strange bend on the left end of the line to them. The next example shows a stronger case. Usually, $\frac{f}{2}N$ points on each side should be discarded. On the other hand, LOESS has the advantage of always staying within the bounds of the data.

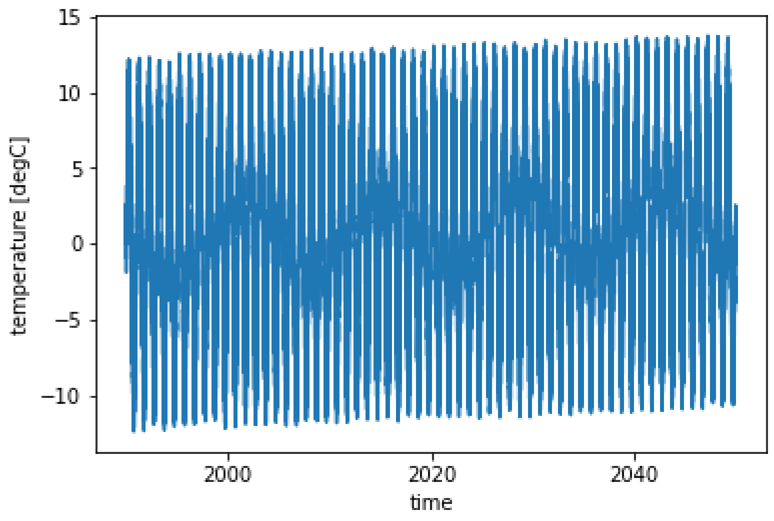
LOESS Detrending

In climate science, it can be used in the detrending process. `xclim` provides `sdba.detrending.LoessDetrend` in order to compute trend with the LOESS smoothing and remove them from timeseries.

First we create some toy data with a sinusoidal annual cycle, random noise and a linear temperature increase.

[3]:
```
time = xr.cftime_range("1990-01-01", "2049-12-31", calendar="noleap")
tas = xr.DataArray(
    (
        10 * np.sin(time.dayofyear * 2 * np.pi / 365)
        + 5 * (np.random.random_sample(time.size) - 0.5)  # Annual variability
        + np.linspace(0, 1.5, num=time.size)  # Random noise
    ),  # 1.5 degC increase in 60 years
    dims=("time",),
    coords={"time": time},
    attrs={"units": "degC"},
    name="temperature",
)
tas.plot()
```

[3]:
```
[<matplotlib.lines.Line2D at 0x7f3d7e2e4130>]
```

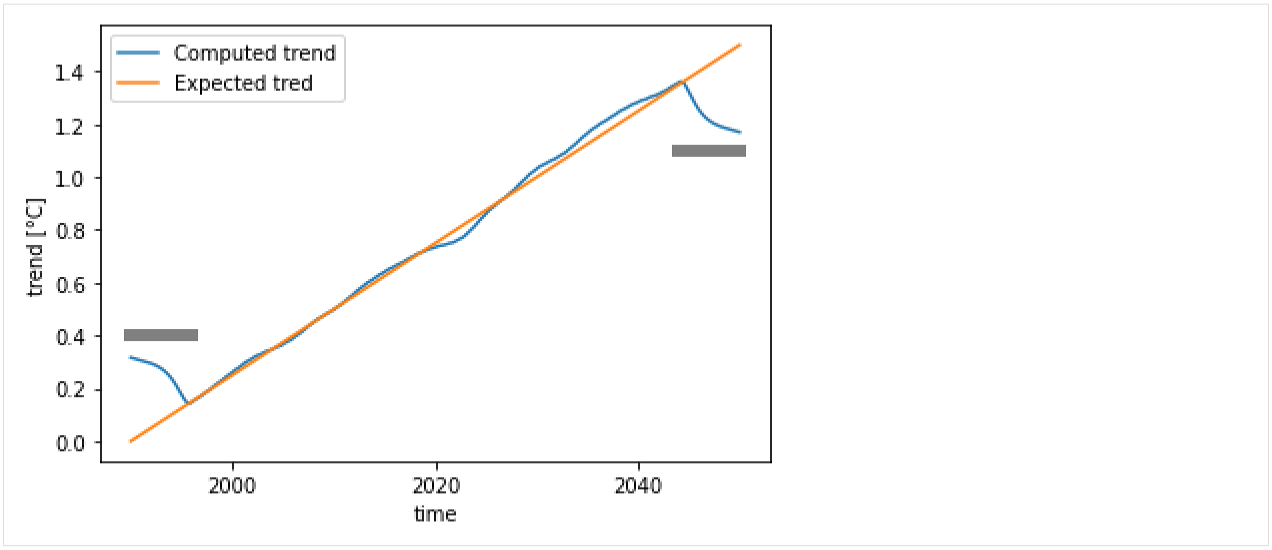


Then we compute the trend on the data. Here, we compute on the whole timeseries (`group='time'`) with the parameters suggested above.

[4]:
```
from xclim.sdba.detrending import LoessDetrend

# Create the detrending object
det = LoessDetrend(group="time", d=0, niter=2, f=0.2)
# Fitting returns a new object and computes the trend.
fit = det.fit(tas)
# Get the detrended series
tas_det = fit.detrend(tas)
```

[5]:
```
fig, ax = plt.subplots()
fit.ds.trend.plot(ax=ax, label="Computed trend")
ax.plot(time, np.linspace(0, 1.5, num=time.size), label="Expected tred")
ax.plot([time[0], time[int(0.1 * time.size)]], [0.4, 0.4], linewidth=6, color="gray")
ax.plot([time[-int(0.1 * time.size)], time[-1]], [1.1, 1.1], linewidth=6, color="gray")
ax.legend()
```

[5]:
```
<matplotlib.legend.Legend at 0x7f3d7d8f31c0>
```



As said earlier, this example shows how the Loess has strong boundary effects. It is recommended to remove the $\frac{f}{2} \cdot N$ outermost points on each side, as shown by the gray bars in the graph above.

3.8.3 Initializing an Adjustment object from a training dataset

For large scale uses, when the training step deserves its own computation and write to disk, or simply when there are multiples `sim` to be adjusted with the same training, it is helpful to be able to instantiate the Adjustment objects from the training dataset itself. This trick relies on a global attribute "adj_params" set on the training dataset.

[6]:
```python
import numpy as np
import xarray as xr

# Create toy data for the example, here fake temperature timeseries
t = xr.cftime_range("2000-01-01", "2030-12-31", freq="D", calendar="noleap")
ref = xr.DataArray(
    (
        -20 * np.cos(2 * np.pi * t.dayofyear / 365)
        + 2 * np.random.random_sample((t.size,))
        + 273.15
        + 0.1 * (t - t[0]).days / 365
    ),  # "warming" of 1K per decade,
    dims=("time",),
    coords={"time": t},
    attrs={"units": "K"},
)
sim = xr.DataArray(
    (
        -18 * np.cos(2 * np.pi * t.dayofyear / 365)
        + 2 * np.random.random_sample((t.size,))
        + 273.15
        + 0.11 * (t - t[0]).days / 365
    ),  # "warming" of 1.1K per decade
    dims=("time",),
```

(continues on next page)

(continued from previous page)

```
    coords={"time": t},
    attrs={"units": "K"},
)

ref = ref.sel(time=slice(None, "2015-01-01"))
hist = sim.sel(time=slice(None, "2015-01-01"))
```

[7]:
```
from xclim.sdba.adjustment import QuantileDeltaMapping

QDM = QuantileDeltaMapping.train(
    ref, hist, nquantiles=15, kind="+", group="time.dayofyear"
)
QDM
```

[7]:
```
QuantileDeltaMapping(group=Grouper(add_dims=[], name='time.dayofyear', window=1), kind='+
↪')
```

The trained QDM exposes the training data in the ds attribute, Here, we will write it to disk, read it back and initialize an new object from it. Notice the adj_params in the dataset, that has the same value as the repr string printed just above. Also, notice the _xclim_adjustment attribute that contains a json string so we can rebuild the adjustment object later.

[8]:
```
QDM.ds
```

[8]:
```
<xarray.Dataset>
Dimensions:    (quantiles: 15, dayofyear: 365)
Coordinates:
  * quantiles  (quantiles) float64 0.03333 0.1 0.1667 ... 0.8333 0.9 0.9667
  * dayofyear  (dayofyear) int64 1 2 3 4 5 6 7 8 ... 359 360 361 362 363 364 365
Data variables:
    af         (dayofyear, quantiles) float64 -2.156 -2.108 ... -1.868 -1.751
    hist_q     (dayofyear, quantiles) float64 255.8 256.1 256.2 ... 257.6 258.0
Attributes:
    group:                time.dayofyear
    group_compute_dims:   ['time']
    group_window:         1
    _xclim_adjustment:    {"py/object": "xclim.sdba.adjustment.QuantileDeltaMa...
    adj_params:           QuantileDeltaMapping(group=Grouper(add_dims=[], name...
```

[9]:
```
QDM.ds.to_netcdf("QDM_training.nc")
ds = xr.open_dataset("QDM_training.nc")
QDM2 = QuantileDeltaMapping.from_dataset(ds)
QDM2
```

[9]:
```
QuantileDeltaMapping(group=Grouper(add_dims=[], name='time.dayofyear', window=1), kind='+
↪')
```

In the case above, creating a full object from the dataset doesn't make the most sense since we are in the same python session, with the "old" object still available. This method effective when we reload the training data in a different python session, say on another computer. **However, take note that there is no retrocompatiblity insurance.** If the QuantileDeltaMapping class was to change in a new xclim version, one would not be able to create the new object from a dataset saved with the old one.

For the case where we stay in the same python session, it is still useful to trigger the dask computations.

For small datasets, that could mean a simple `QDM.ds.load()`, but sometimes even the training data is too large to be full loaded in memory. In that case, we could also do:

```
[10]: QDM.ds.to_netcdf("QDM_training2.nc")
ds = xr.open_dataset("QDM_training2.nc")
ds.attrs["title"] = "This is the dataset, but read from disk."
QDM.set_dataset(ds)
QDM.ds
```

```
[10]: <xarray.Dataset>
Dimensions:    (quantiles: 15, dayofyear: 365)
Coordinates:
  * quantiles  (quantiles) float64 0.03333 0.1 0.1667 ... 0.8333 0.9 0.9667
  * dayofyear  (dayofyear) int64 1 2 3 4 5 6 7 8 ... 359 360 361 362 363 364 365
Data variables:
    af         (dayofyear, quantiles) float64 ...
    hist_q     (dayofyear, quantiles) float64 ...
Attributes:
    group:               time.dayofyear
    group_compute_dims:  time
    group_window:        1
    _xclim_adjustment:   {"py/object": "xclim.sdba.adjustment.QuantileDeltaMa...
    adj_params:          QuantileDeltaMapping(group=Grouper(add_dims=[], name...
    title:               This is the dataset, but read from disk.
```

```
[11]: QDM2.adjust(sim)
```

```
[11]: <xarray.DataArray 'scen' (time: 11315)>
array([254.66138872, 253.49410061, 253.3465842 , ..., 256.93444439,
       256.98717378, 258.51507099])
Coordinates:
  * time     (time) object 2000-01-01 00:00:00 ... 2030-12-31 00:00:00
Attributes:
    units:             K
    history:           [2022-06-18 02:35:03] : Bias-adjusted with QuantileDelt...
    bias_adjustment:   QuantileDeltaMapping(group=Grouper(add_dims=[], name='t...
```

3.8.4 Retrieving extra output diagnostics

To fully understand what is happening during the bias-adjustment process, sdba can output *diagnostic* variables, giving more visibility to what the adjustment is doing behind the scene. This behaviour, a `verbose` option, is controlled by the `sdba_extra_output` option, set with `xclim.set_options`. When `True`, `train` calls are instructed to include additional variables to the training datasets. In addition, the `adjust` calls will always output a dataset, with `scen` and, depending on the algorithm, other diagnostics variables. See the documentation of each `Adjustment` objects to see what extra variables are available.

For the moment, this feature is still in construction and only a few `Adjustment` actually provide extra outputs. Please open issues on the Github repo if you have needs or ideas of interesting diagnostic variables.

For example, `QDM.adjust` adds `sim_q`, which gives the quantile of each element of `sim` within its group.

```
[12]: from xclim import set_options

with set_options(sdba_extra_output=True):
```

(continues on next page)

(continued from previous page)

```
    QDM = QuantileDeltaMapping.train(
        ref, hist, nquantiles=15, kind="+", group="time.dayofyear"
    )
    out = QDM.adjust(sim)

out.sim_q
```

[12]:
```
<xarray.DataArray 'sim_q' (time: 11315)>
array([0.25806452, 0.03225806, 0.03225806, ..., 0.96774194, 0.83870968,
       1.        ])
Coordinates:
  * time     (time) object 2000-01-01 00:00:00 ... 2030-12-31 00:00:00
Attributes:
    group:               time.dayofyear
    group_compute_dims:  time
    group_window:        1
    long_name:           Group-wise quantiles of `sim`.
```

3.8.5 Moving window for adjustments

Some Adjustment methods require that the adjusted data (`sim`) be of the same length (same number of points) than the training data (`ref` and `hist`). This requirements often ensure conservation of statistical properties and a better representation of the climate change signal over the long adjusted timeseries.

In opposition to a conventionnal "rolling window", here it is the *years* that are the base units of the window, not the elements themselves. xclim implements `sdba.construct_moving_yearly_window` and `sdba.unpack_moving_yearly_window` to manipulate data in that goal. The "construct" function cuts the data in overlapping windows of a certain length (in years) and stacks them along a new `"movingdim"` dimension, alike to xarray's `da.rolling(time=win).construct('movingdim')`, but with yearly steps. The step between each window can also be controlled. This argument is an indicator of how many years overlap between each window. With a value of 1 (the default), a window will have `window - 1` years overlapping with the previous one. `step = window` will result in no overlap at all.

By default, the result is chunked along this `'movingdim'` dimension. For this reason, the method is expected to be more computationally efficient (when using `dask`) than looping over the windows.

Note that this results in two restrictions:

1. The constructed array has the same "time" axis for all windows. This is a problem if the actual *year* is of importance for the adjustment, but this is not the case for any of xclim's current adjustment methods.
2. The input timeseries must be in a calendar with uniform year lengths. For daily data, this means only the "360_day", "noleap" and "all_leap" calendars are supported.

The "unpack" function does the opposite : it concatenates the windows together to recreate the original timeseries. The time points that are not part of a window will not appear in the reconstructed timeseries. If `append_ends` is True, the reconstructed timeseries will go from the first time point of the first window to the last time point of the last window. In the middle, the central `step` years are kept from each window. If `append_ends` is False, only the central `step` years are kept from each window. Which means the final timeseries has `(window - step) / 2` years missing on either side, with the extra year missing on the right in case of an odd `(window - step)`. We are missing data, but the contribution from each window is equal.

Here, as`ref` and `hist` cover 15 years, we will use a window of 15 on sim. With a step of 2, this means the first window goes from 2000 to 2014 (inclusive). The last window goes from 2016 to 2030. `window - step`

= 13, so 6 years will be missing at the beginning of the final `scen` and 7 years at the end.

[13]:
```
QDM = QuantileDeltaMapping.train(
    ref, hist, nquantiles=15, kind="+", group="time.dayofyear"
)

scen_nowin = QDM.adjust(sim)
```

[14]:
```
sim
```

[14]:
```
<xarray.DataArray (time: 11315)>
array([257.118449, 255.360641, 255.429816, ..., 259.578762, 259.275183,
       260.266003])
Coordinates:
  * time     (time) object 2000-01-01 00:00:00 ... 2030-12-31 00:00:00
Attributes:
    units:    K
```

[15]:
```
from xclim.sdba import construct_moving_yearly_window, unpack_moving_yearly_window

sim_win = construct_moving_yearly_window(sim, window=15, step=2)
sim_win
```

[15]:
```
<xarray.DataArray (movingwin: 9, time: 5475)>
array([[257.1184495 , 255.36064145, 255.42981576, ..., 256.95060691,
        257.44911234, 257.02462747],
       [255.72034502, 255.90052092, 257.13951078, ..., 257.93005318,
        257.66474239, 258.34192162],
       [256.30745872, 257.53165621, 257.10270258, ..., 258.76808972,
        258.83187767, 257.74206442],
       ...,
       [258.0653123 , 257.74234015, 257.85017198, ..., 258.62620633,
        259.8371446 , 258.78171416],
       [258.30373866, 257.77902395, 258.56627058, ..., 258.36899253,
        259.04597558, 259.20341891],
       [257.87602306, 257.65165498, 257.60703721, ..., 259.57876233,
        259.27518344, 260.26600327]])
Coordinates:
  * movingwin  (movingwin) object 2000-01-01 00:00:00 ... 2016-01-01 00:00:00
  * time       (time) object 2000-01-01 00:00:00 ... 2014-12-31 00:00:00
Attributes:
    units:    K
```

Here, we retrieve the full timeseries.

[16]:
```
scen_win = unpack_moving_yearly_window(QDM.adjust(sim_win), append_ends=True)
scen_win
```

[16]:
```
<xarray.DataArray 'scen' (time: 11315)>
array([254.8028273 , 253.49410061, 253.3465842 , ..., 257.08293134,
       257.13066266, 258.51507099])
Coordinates:
  * time     (time) object 2000-01-01 00:00:00 ... 2030-12-31 00:00:00
Attributes:
    units:             K
```

(continues on next page)

(continued from previous page)

```
    history:           [2022-06-18 02:35:08] : Bias-adjusted with QuantileDelt...
    bias_adjustment:  QuantileDeltaMapping(group=Grouper(add_dims=[], name='t...
```

Whereas here, we have gaps at the edges.

```
[17]: scen_win = unpack_moving_yearly_window(QDM.adjust(sim_win), append_ends=False)
scen_win
```

```
[17]: <xarray.DataArray 'scen' (time: 6570)>
array([255.0757119 , 254.34256973, 254.4739955 , ..., 256.51318461,
       256.39212504, 257.12619826])
Coordinates:
  * time     (time) object 2006-01-01 00:00:00 ... 2023-12-31 00:00:00
Attributes:
    units:             K
    history:           [2022-06-18 02:35:08] : Bias-adjusted with QuantileDelt...
    bias_adjustment:  QuantileDeltaMapping(group=Grouper(add_dims=[], name='t...
```

Here is another short example, with an uneven number of years. Here `sim` goes from 2000 to 2029 (30 years instead of 31). With a step of 2 and a window of 15, the first window goes again from 2000 to 2014, but the last one is now from 2014 to 2028. The next window would be 2016-2030, but that last year doesn't exist.

```
[18]: sim_win = construct_moving_yearly_window(
    sim.sel(time=slice("2000", "2029")), window=15, step=2
)
sim_win
```

```
[18]: <xarray.DataArray (movingwin: 8, time: 5475)>
array([[257.1184495 , 255.36064145, 255.42981576, ..., 256.95060691,
        257.44911234, 257.02462747],
       [255.72034502, 255.90052092, 257.13951078, ..., 257.93005318,
        257.66474239, 258.34192162],
       [256.30745872, 257.53165621, 257.10270258, ..., 258.76808972,
        258.83187767, 257.74206442],
       ...,
       [257.03621442, 256.79116107, 257.77180978, ..., 259.30800236,
        258.16523804, 259.66852812],
       [258.0653123 , 257.74234015, 257.85017198, ..., 258.62620633,
        259.8371446 , 258.78171416],
       [258.30373866, 257.77902395, 258.56627058, ..., 258.36899253,
        259.04597558, 259.20341891]])
Coordinates:
  * movingwin  (movingwin) object 2000-01-01 00:00:00 ... 2014-01-01 00:00:00
  * time       (time) object 2000-01-01 00:00:00 ... 2014-12-31 00:00:00
Attributes:
    units:    K
```

Here, we don't recover the full timeseries, even when we append the ends, because 2029 is not part of a window.

```
[19]: sim2 = unpack_moving_yearly_window(sim_win, append_ends=True)
sim2
```

[19]:
```
<xarray.DataArray (time: 10585)>
array([257.1184495 , 255.36064145, 255.42981576, ..., 258.36899253,
       259.04597558, 259.20341891])
Coordinates:
  * time     (time) object 2000-01-01 00:00:00 ... 2028-12-31 00:00:00
Attributes:
    units:    K
```

Without appending the ends, the final timeseries is from 2006 to 2021, 6 years missing at the beginning, like last time and **8** years missing at the end.

[20]:
```python
sim2 = unpack_moving_yearly_window(sim_win, append_ends=False)
sim2
```

[20]:
```
<xarray.DataArray (time: 5840)>
array([257.26499483, 255.90110851, 256.90021711, ..., 259.49845773,
       259.47263566, 259.42614294])
Coordinates:
  * time     (time) object 2006-01-01 00:00:00 ... 2021-12-31 00:00:00
Attributes:
    units:    K
```

3.8.6 Full example: Multivariate adjustment in the additive space

The following example shows a complete bias-adjustment workflow using the `PrincipalComponents` method in a multi-variate configuration. Moreover, it uses the trick showed by Alavoine et Grenier (2022) to transform "multiplicative" variable to the "additive" space using log and logit transformations. This way, we can perform multi-variate adjustment with variables that couldn't be used in the same *kind* of adjustment, like "tas" and "hurs".

We will transform the variables that need it to the additive space, adding some jitter in the process to avoid $log(0)$ situations. Then, we will stack the different variables into a single `DataArray`, allowing us to use to use `PrincipalComponents` in a multi-variate way. Following the PCA, a simple quantile-mapping method is used, both adjustment acting on the residuals, while the mean of the simulated trend is adjusted on its own. Each step will be explained.

First, open the data, convert the calendar and the units. Because we will perform adjustments on "dayofyear" groups (with a window), keeping standard calendars results in a extra "dayofyear" with only a quarter of the data. It's usual to transform to a "noleap" calendar, which drops the 29th of February, as it only has a small impact on the data.

[21]:
```python
import xclim.sdba as sdba
from xclim.core.calendar import convert_calendar
from xclim.core.units import convert_units_to
from xclim.testing import open_dataset

group = sdba.Grouper("time.dayofyear", window=31)

dref = convert_calendar(open_dataset("sdba/ahccd_1950-2013.nc"), "noleap").sel(
    time=slice("1981", "2010")
)
dsim = open_dataset("sdba/CanESM2_1950-2100.nc")

```
(continues on next page)

(continued from previous page)

```
dref = dref.assign(
    tasmax=convert_units_to(dref.tasmax, "K"),
)
dsim = dsim.assign(pr=convert_units_to(dsim.pr, "mm/d"))
```

1. Jitter, additive space transformation and variable stacking

Here, `tasmax` is already ready to be adjusted in an additive way, because all data points are far from the physical zero (0 K). This is not the case for `pr`, which is why we want to transform that variable to the additive space, to avoid splitting our workflow in two. For `pr` the "log" transformation is simply:

$$pr' = \ln(pr - b)$$

where b is the lower bound, here 0 mm/d. However, we could have exact zeros (0 mm/d) in the datasets, which will translate into $-\infty$. To avoid this, we simply replace the smallest values by a random distribution of very small, but not problematic, values. In the following, all values below 0.1 mm/d are replace by a uniform random distribution of values within the range (0, 0.1) mm/d (bounds excluded).

Finally, the variables are stacked together into a single DataAray.

[22]:
```
dref_as = dref.assign(
    pr=sdba.processing.to_additive_space(
        sdba.processing.jitter(dref.pr, lower="0.1 mm/d", minimum="0 mm/d"),
        lower_bound="0 mm/d",
        trans="log",
    )
)
ref = sdba.stack_variables(dref_as)

dsim_as = dsim.assign(
    pr=sdba.processing.to_additive_space(
        sdba.processing.jitter(dsim.pr, lower="0.1 mm/d", minimum="0 mm/d"),
        lower_bound="0 mm/d",
        trans="log",
    )
)
sim = sdba.stack_variables(dsim_as)
sim
```

[22]:
```
<xarray.DataArray 'multivariate' (multivar: 2, time: 55115, location: 3)>
array([[[ 2.4951415e-01, -8.2575524e-01,  2.4951415e-01],
        [ 2.6499695e-01, -4.1112199e-01,  2.6499695e-01],
        [-1.9535363e-01, -2.5183392e+00, -1.9535363e-01],
        ...,
        [ 3.2132244e+00, -2.2834629e-01,  3.2132244e+00],
        [ 1.6713389e+00,  1.7489431e+00,  1.6713389e+00],
        [ 7.5195438e-01,  2.4332016e+00,  7.5195438e-01]],

       [[ 2.7815024e+02,  2.7754898e+02,  2.7815024e+02],
        [ 2.8335815e+02,  2.7690921e+02,  2.8335815e+02],
        [ 2.8153192e+02,  2.7668036e+02,  2.8153192e+02],
        ...,
```

(continues on next page)

(continued from previous page)

```
        [ 2.8901334e+02,  2.8192789e+02,  2.8901334e+02],
        [ 2.8510699e+02,  2.8142294e+02,  2.8510699e+02],
        [ 2.8404471e+02,  2.8160156e+02,  2.8404471e+02]]], dtype=float32)
Coordinates:
  * time      (time) object 1950-01-01 00:00:00 ... 2100-12-31 00:00:00
    lat       (location) float64 49.1 67.8 48.8
    lon       (location) float64 -123.1 -115.1 -78.2
  * location  (location) object 'Vancouver' 'Kugluktuk' 'Amos'
  * multivar  (multivar) <U6 'pr' 'tasmax'
Attributes:
    institution:                     CanESM2
    institute_id:                    CCCma
    experiment_id:                   rcp85
    source:                          CanESM2 2010 atmosphere: CanAM4 (AGCM15i...
    model_id:                        CanESM2
    forcing:                         GHG,Oz,SA,BC,OC,LU,Sl (GHG includes CO2,...
    parent_experiment_id:            historical
    parent_experiment_rip:           r1i1p1
    branch_time:                     56940.0
    contact:                         cccma_info@ec.gc.ca
    references:                      http://www.cccma.ec.gc.ca/models
    initialization_method:           1
    physics_version:                 1
    tracking_id:                     17560481-e4c5-43c9-bc3f-950732f21588
    branch_time_YMDH:                2006:01:01:00
    CCCma_runid:                     IDR
    CCCma_parent_runid:              IGM
    CCCma_data_licence:              1) GRANT OF LICENCE - The Government of ...
    product:                         output
    experiment:                      RCP8.5
    frequency:                       day
    creation_date:                   2011-04-10T11:24:15Z
    history:                         2021-04-23T12:00:00: Extraction of times...
    Conventions:                     CF-1.4
    project_id:                      CMIP5
    table_id:                        Table day (28 March 2011) f9d6cfec5981bb...
    title:                           Test dataset for xclim.sdba - model data
    parent_experiment:               historical
    modeling_realm:                  atmos
    realization:                     1
    cmor_version:                    2.5.4
    DODS_EXTRA.Unlimited_Dimension:  time
    description:                     Extracted from CMIP5 CanESM2 hist+rcp85 ...
    units:
```

2. Get residuals and trends

The adjustment will be performed on residuals only. The adjusted timeseries `sim` will be detrended with the LOESS routine described above. Because of the short length of `ref` and `hist` and the potential boundary effects of using LOESS with them, we compute the 30-year mean. In other words, instead of *detrending* we are *normalizing* those inputs.

While the residuals are adjusted with `PrincipalComponents` and `EmpiricalQuantileMapping`, the trend of `sim` still needs to be offset according to the means of `ref` and `hist`. This is similar to what `DetrendedQuantileMapping` does. The offset step could have been done on the trend itself or at the end on `scen`, it doesn't really matter. We do it here because it keeps it close to where the `scaling` is computed.

[23]:
```
ref_res, ref_norm = sdba.processing.normalize(ref, group=group, kind="+")
hist_res, hist_norm = sdba.processing.normalize(
    sim.sel(time=slice("1981", "2010")), group=group, kind="+"
)
scaling = sdba.utils.get_correction(hist_norm, ref_norm, kind="+")
```

[24]:
```
sim_scaled = sdba.utils.apply_correction(
    sim, sdba.utils.broadcast(scaling, sim, group=group), kind="+"
)

loess = sdba.detrending.LoessDetrend(group=group, f=0.2, d=0, kind="+", niter=1)
simfit = loess.fit(sim_scaled)
sim_res = simfit.detrend(sim_scaled)
```

3. Adjustments

Following, Alavoine et Grenier (2022), we decided to perform the multivariate Principal Components adjustment first and then re-adjust with the simple quantile-mapping.

[25]:
```
PCA = sdba.adjustment.PrincipalComponents.train(
    ref_res, hist_res, group=group, crd_dim="multivar", best_orientation="simple"
)

scen1_res = PCA.adjust(sim_res)
```

[26]:
```
EQM = sdba.adjustment.EmpiricalQuantileMapping.train(
    ref_res,
    scen1_res.sel(time=slice("1981", "2010")),
    group=group,
    nquantiles=50,
    kind="+",
)

scen2_res = EQM.adjust(scen1_res, interp="linear", extrapolation="constant")
```

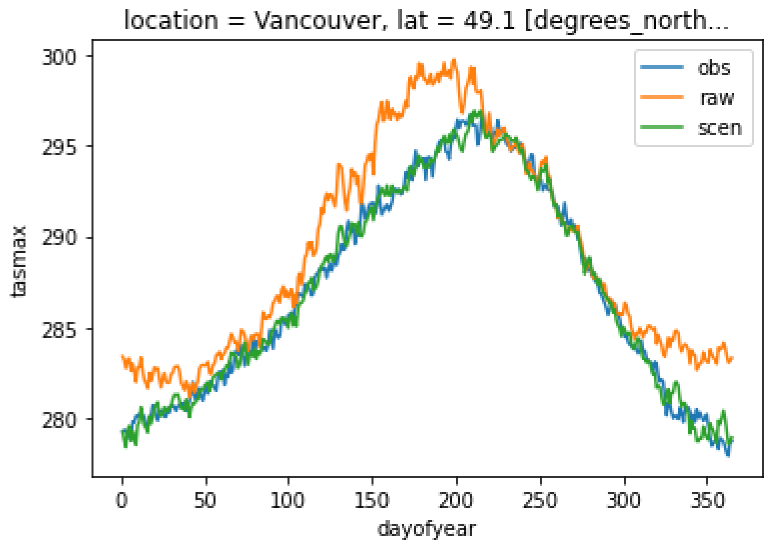
4. Re-trend and transform back to the physical space

Add back the trend (which includes the scaling), unstack the variables to a dataset and transform `pr` back to the physical space. All functions have conserved and handled the attributes, so we don't need to repeat the additive space bounds. The annual cycle of both variables on the reference period in Vancouver is plotted to confirm the adjustment add a positive effect.

[27]:
```
scen = simfit.retrend(scen2_res)
dscen_as = sdba.unstack_variables(scen)
dscen = dscen_as.assign(pr=sdba.processing.from_additive_space(dscen_as.pr))
```

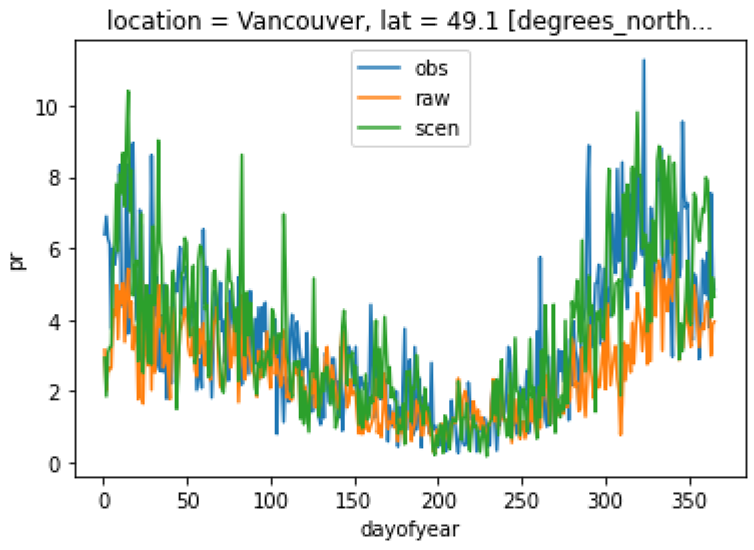
[28]:
```
dref.tasmax.sel(time=slice("1981", "2010"), location="Vancouver").groupby(
    "time.dayofyear"
).mean().plot(label="obs")
dsim.tasmax.sel(time=slice("1981", "2010"), location="Vancouver").groupby(
    "time.dayofyear"
).mean().plot(label="raw")
dscen.tasmax.sel(time=slice("1981", "2010"), location="Vancouver").groupby(
    "time.dayofyear"
).mean().plot(label="scen")
plt.legend()
```

[28]:
```
<matplotlib.legend.Legend at 0x7f3d7d8e2fa0>
```



[29]:
```
dref.pr.sel(time=slice("1981", "2010"), location="Vancouver").groupby(
    "time.dayofyear"
).mean().plot(label="obs")
dsim.pr.sel(time=slice("1981", "2010"), location="Vancouver").groupby(
    "time.dayofyear"
).mean().plot(label="raw")
dscen.pr.sel(time=slice("1981", "2010"), location="Vancouver").groupby(
    "time.dayofyear"
).mean().plot(label="scen")
plt.legend()
```

[29]:
```
<matplotlib.legend.Legend at 0x7f3d7652d190>
```



3.8.7 Tests for sdba

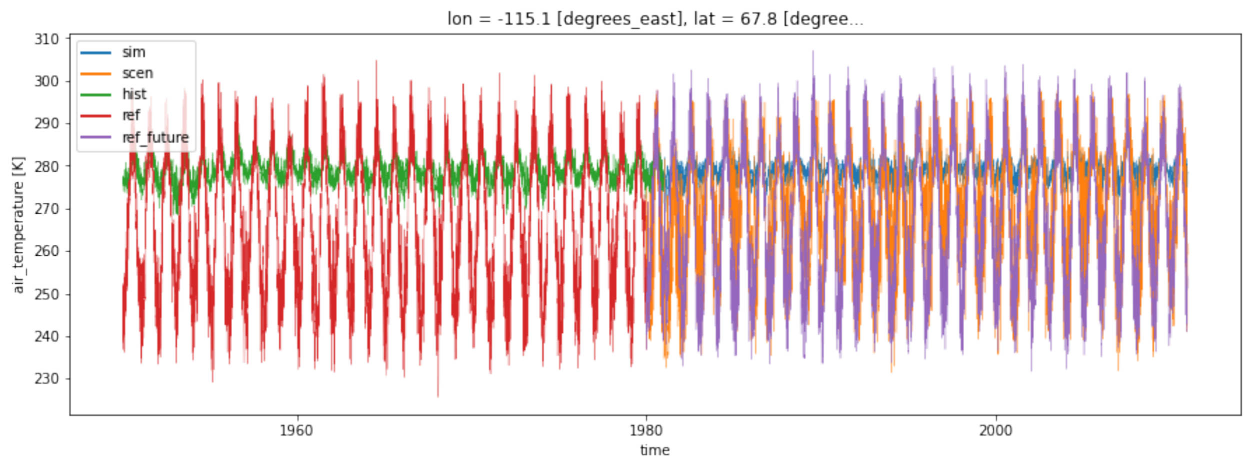
It can be useful to perform diagnostic tests on adjusted simulations to assess if the bias correction method is working properly or to compare two different bias correction techniques.

A diagnostic test includes calculations of a property (mean, 20-year return value, annual cycle amplitude, . . .) on the simulation and on the scenario (adjusted simulation), then a measure (bias, relative bias, ratio, . . .) of the difference. The property collapse the time dimension of the simulation/scenario and returns one value by grid point.

[30]:
```python
from matplotlib import pyplot as plt

import xclim as xc
from xclim import sdba
from xclim.testing import open_dataset

# load test data
hist = open_dataset("sdba/CanESM2_1950-2100.nc").sel(time=slice("1950", "1980")).tasmax
ref = open_dataset("sdba/nrcan_1950-2013.nc").sel(time=slice("1950", "1980")).tasmax
sim = (
    open_dataset("sdba/CanESM2_1950-2100.nc").sel(time=slice("1980", "2010")).tasmax
)  # biased

# learn the bias in historical simulation compared to reference
QM = sdba.EmpiricalQuantileMapping.train(
    ref, hist, nquantiles=50, group="time", kind="+"
)

# correct the bias in the future
scen = QM.adjust(sim, extrapolation="constant", interp="nearest")
ref_future = (
    open_dataset("sdba/nrcan_1950-2013.nc").sel(time=slice("1980", "2010")).tasmax
```
(continues on next page)

(continued from previous page)

```
)  # truth

plt.figure(figsize=(15, 5))
lw = 0.3
sim.isel(location=1).plot(label="sim", linewidth=lw)
scen.isel(location=1).plot(label="scen", linewidth=lw)
hist.isel(location=1).plot(label="hist", linewidth=lw)
ref.isel(location=1).plot(label="ref", linewidth=lw)
ref_future.isel(location=1).plot(label="ref_future", linewidth=lw)
leg = plt.legend()
for legobj in leg.legendHandles:
    legobj.set_linewidth(2.0)
```



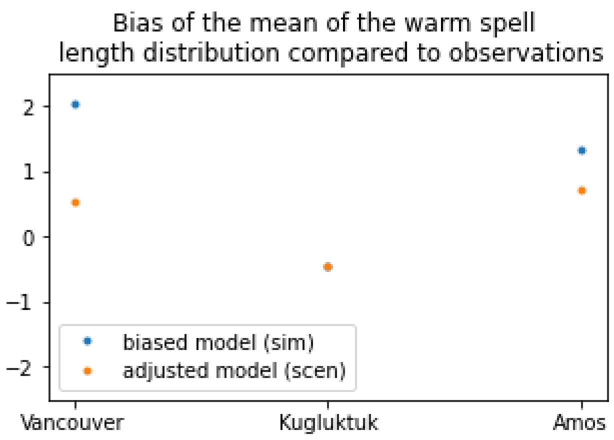
[31]:
```
# calculate the mean warm Spell Length Distribution
sim_prop = xc.sdba.properties.spell_length_distribution(
    da=sim, thresh="28 degC", op=">", stat="mean", group="time"
)


scen_prop = xc.sdba.properties.spell_length_distribution(
    da=scen, thresh="28 degC", op=">", stat="mean", group="time"
)

ref_prop = xc.sdba.properties.spell_length_distribution(
    da=ref_future, thresh="28 degC", op=">", stat="mean", group="time"
)
# measure the difference between the prediction and the reference with an absolute bias␣
↪of the properties
measure_sim = xc.sdba.measures.bias(sim_prop, ref_prop)
measure_scen = xc.sdba.measures.bias(scen_prop, ref_prop)

plt.figure(figsize=(5, 3))
plt.plot(measure_sim.location, measure_sim.values, ".", label="biased model (sim)")
plt.plot(measure_scen.location, measure_scen.values, ".", label="adjusted model (scen)")
plt.title(
    "Bias of the mean of the warm spell \n length distribution compared to observations"
```

(continues on next page)

(continued from previous page)

```
)
plt.legend()
plt.ylim(-2.5, 2.5)
```

[31]:
```
(-2.5, 2.5)
```



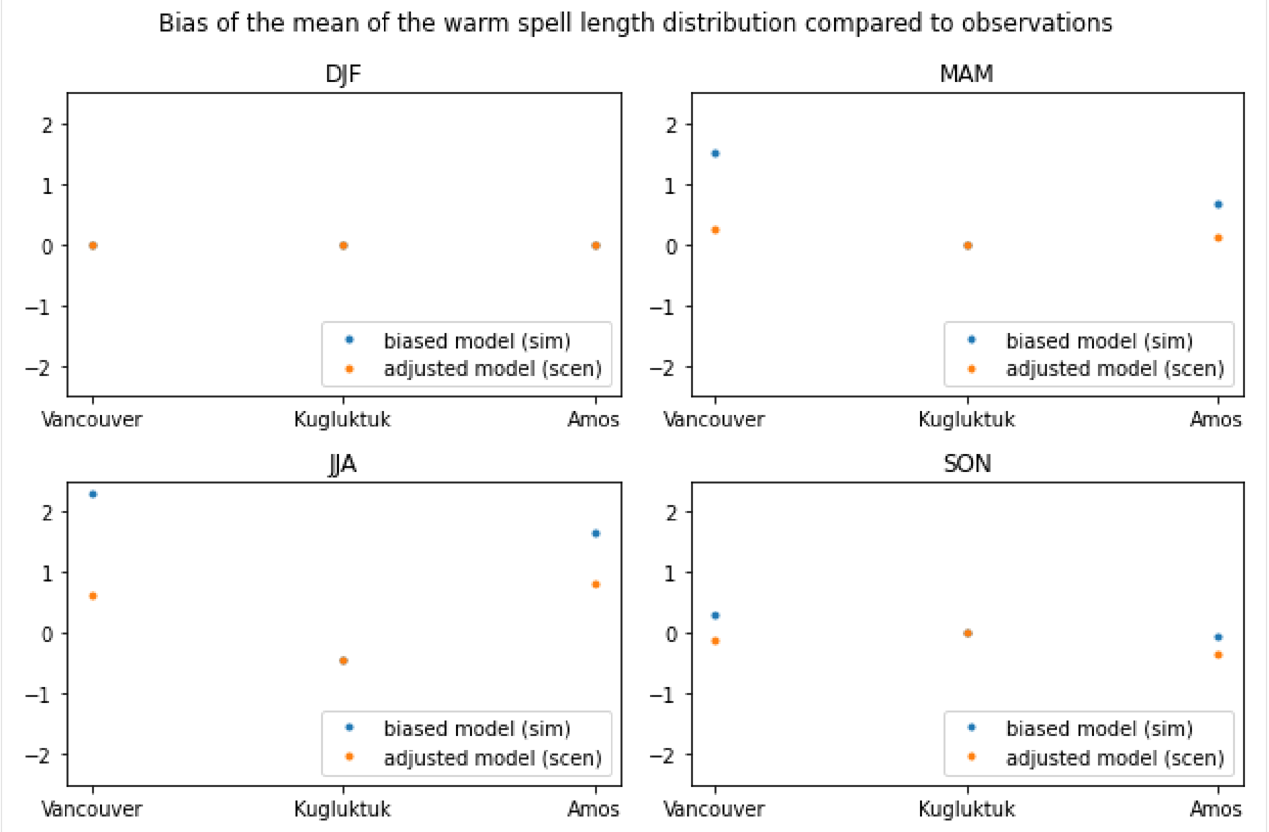
It is possible the change the 'group' of the property from 'time' to 'time.season' or 'time.month'. This will return 4 or 12 values per grid point, respectively.

[32]:
```
# calculate the mean warm Spell Length Distribution
sim_prop = xc.sdba.properties.spell_length_distribution(
    da=sim, thresh="28 degC", op=">", stat="mean", group="time.season"
)

scen_prop = xc.sdba.properties.spell_length_distribution(
    da=scen, thresh="28 degC", op=">", stat="mean", group="time.season"
)

ref_prop = xc.sdba.properties.spell_length_distribution(
    da=ref_future, thresh="28 degC", op=">", stat="mean", group="time.season"
)
# measure the difference between the prediction and the reference with an absolute bias␣
↪the properties
measure_sim = xc.sdba.measures.bias(sim_prop, ref_prop)
measure_scen = xc.sdba.measures.bias(scen_prop, ref_prop)

fig, axs = plt.subplots(2, 2, figsize=(9, 6))
axs = axs.ravel()
for i in range(4):
    axs[i].plot(
        measure_sim.location, measure_sim.values[:, i], ".", label="biased model (sim)"
    )
    axs[i].plot(
        measure_scen.location,
        measure_scen.isel(season=i).values,
        ".",
        label="adjusted model (scen)",
    )
```

(continues on next page)

(continued from previous page)

```
    axs[i].set_title(measure_scen.season.values[i])
    axs[i].legend(loc="lower right")
    axs[i].set_ylim(-2.5, 2.5)
fig.suptitle(
    "Bias of the mean of the warm spell length distribution compared to observations"
)
plt.tight_layout()
```



3.9 Spatial Analogues examples

xclim provides the `xc.analog` module that allows the finding of spatial analogues. Spatial analogues are maps showing which areas have a present-day climate that is analogous to the future climate of a given place. This type of map can be useful for climate adaptation to see how well regions are coping today under specific climate conditions. For example, officials from a city located in a temperate region that may be expecting more heatwaves in the future can learn from the experience of another city where heatwaves are a common occurrence, leading to more proactive intervention plans to better deal with new climate conditions.

Spatial analogues are estimated by comparing the distribution of climate indices computed at the target location over the future period with the distribution of the same climate indices computed over a reference period for multiple candidate regions.

[1]:
```
import matplotlib.pyplot as plt

from xclim import analog
```

(continues on next page)

(continued from previous page)

```
from xclim.core.calendar import convert_calendar
from xclim.testing import open_dataset
```

3.9.1 Input data

The "target" input of the computation is a collection of indices over a given location and for a given time period. Here we have three indices computed on bias-adjusted daily simulation data from the CanESM2 model, as made available through the CMIP5 project. We chose to look at the climate of Chibougamau, a small city in northern Québec, for the 2041-2070 period.

[2]:
```
sim = open_dataset(
    "SpatialAnalogs/CanESM2_ScenGen_Chibougamau_2041-2070.nc",
    branch="spatial-analogs-nb",
    decode_timedelta=False,
)
sim
```

[2]:
```
<xarray.Dataset>
Dimensions:                            (time: 30)
Coordinates:
  * time                               (time) object 2041-01-01 00:00:00 ...
    lon                                float32 ...
    lat                                float32 ...
Data variables:
    tg_mean                            (time) float32 ...
    growing_season_length              (time) float32 ...
    max_n_day_precipitation_amount_n_5  (time) float32 ...
Attributes:
    Conventions:      CF-1.5
    title:            Future climate of Chibougamau, QC - Bias-adjusted data f...
    history:          2011-04-13T23:04:41Z CMOR rewrote data to comply with CF...
    institution:      CCCma (Canadian Centre for Climate Modelling and Analysi...
    source:           CanESM2 2010 atmosphere: CanAM4 (AGCM15i, T63L35) ocean:...
    redistribution:   Redistribution prohibited. For internal use only.
```

The goal is to find regions where the present climate is similar to that simulated future climate. We call "candidates" the dataset that contains the present-day indices. Here we use gridded observations provided by NRCAN. This is the same data that was used as a reference for the bias-adjustment of the target simulation, which is essential to ensure the comparison holds.

A good test to see if the data is appropriate for computing spatial analog is the so-called "self-analog" test. It consists in computing the analogs using the same time period on both the target and the candidates. The test passes if the best analog is the same point as the target. Some authors have found that in some cases, a second bias-adjustment over the indices is needed to ensure that the data passes this test (see Grenier et al. (2019)). However, in this introductory notebook, we can't run this test and will simply assume the data is coherent.

[3]:
```
obs = open_dataset(
    "SpatialAnalogs/NRCAN_SECan_1981-2010.nc",
    branch="spatial-analogs-nb",
    decode_timedelta=False,
```

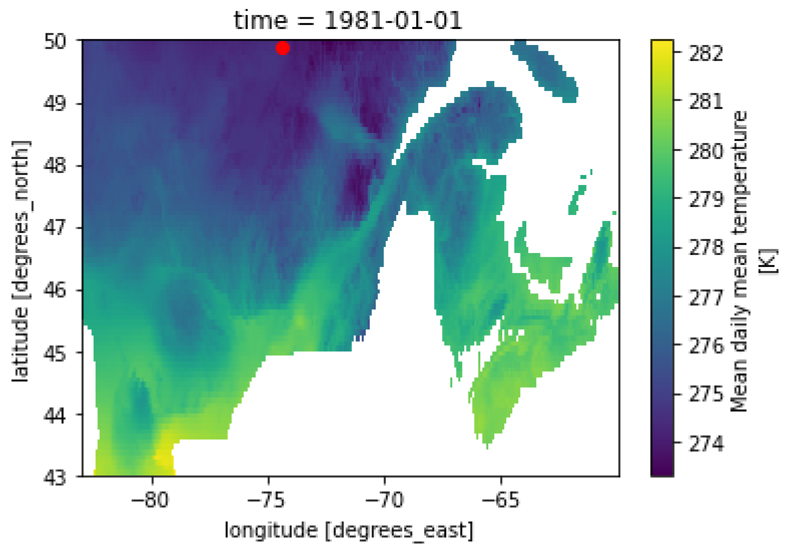
(continues on next page)

(continued from previous page)

```
)
obs
```

[3]:
```
<xarray.Dataset>
Dimensions:                              (time: 30, lon: 276, lat: 84)
Coordinates:
  * time                                 (time) datetime64[ns] 1981-01-01 ... ...
  * lon                                  (lon) float32 -82.96 -82.88 ... -60.04
  * lat                                  (lat) float32 49.96 49.88 ... 43.04
Data variables:
    tg_mean                              (time, lat, lon) float32 ...
    growing_season_length                (time, lat, lon) float32 ...
    max_n_day_precipitation_amount_n_5   (time, lat, lon) float32 ...
Attributes:
    Conventions:      CF-1.5
    title:            NRCAN Gridded observations over southern Quebec
    history:          2012-10-22T13:14:41: Convert from original format to Net...
    institution:      NRCAN
    source:           ANUSPLIN
    redistribution:   Redistribution policy unknown. For internal use only.
```

[4]:
```
obs.tg_mean.isel(time=0).plot()
plt.plot(sim.lon, sim.lat, "ro");   # Plot a point over chibougamau
```



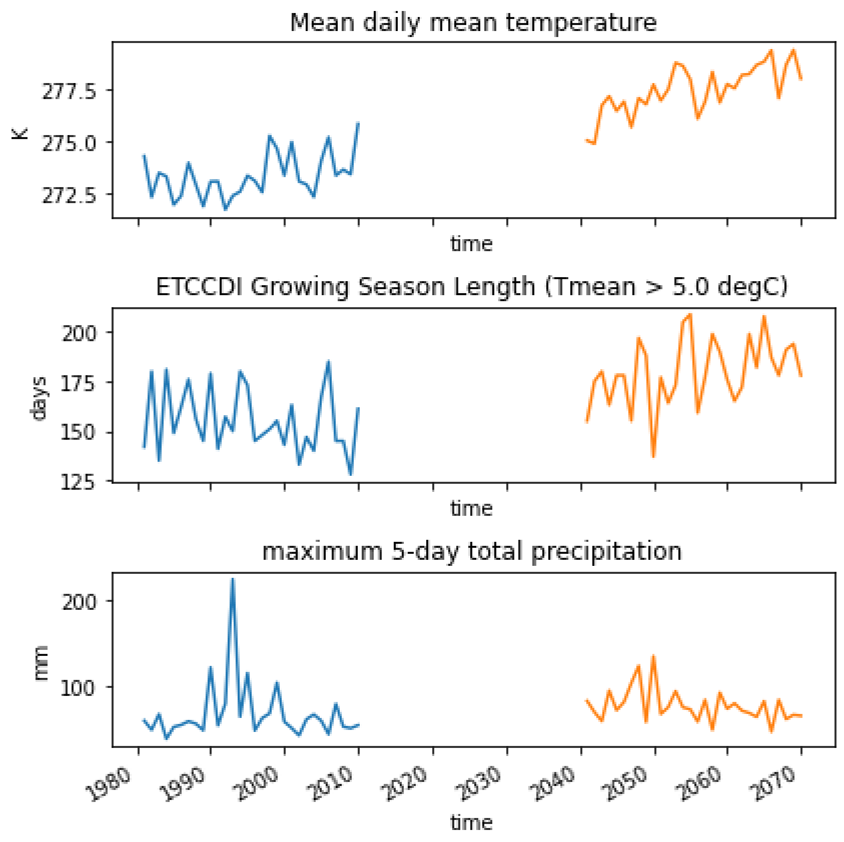
Let's plot the timeseries over Chibougamau for both periods to get an idea of the climate change between the two periods. For the purpose of the plot, we'll need to convert the calendar of the data as the simulation uses a "noleap" calendar.

[5]:
```
fig, axs = plt.subplots(nrows=3, figsize=(6, 6), sharex=True)
sim_std = convert_calendar(sim, "default")
obs_chibou = obs.sel(lat=sim.lat, lon=sim.lon, method="nearest")

for ax, var in zip(axs, obs_chibou.data_vars.keys()):
    obs_chibou[var].plot(ax=ax, label="Observation")
```

(continues on next page)

(continued from previous page)

```
    sim_std[var].plot(ax=ax, label="Simulation")
    ax.set_title(obs_chibou[var].long_name)
    ax.set_ylabel(obs_chibou[var].units)
fig.tight_layout()
```



All the work is encapsulated in the `xclim.analog.spatial_analogs` function. By default, the function expects that the distribution to be analyzed is along the "time" dimension, like in our case. Inputs are datasets of indices, the target and the candidates should have the same indices and at least the `time` variable in common. Normal xarray broadcasting rules apply for the other dimensions.

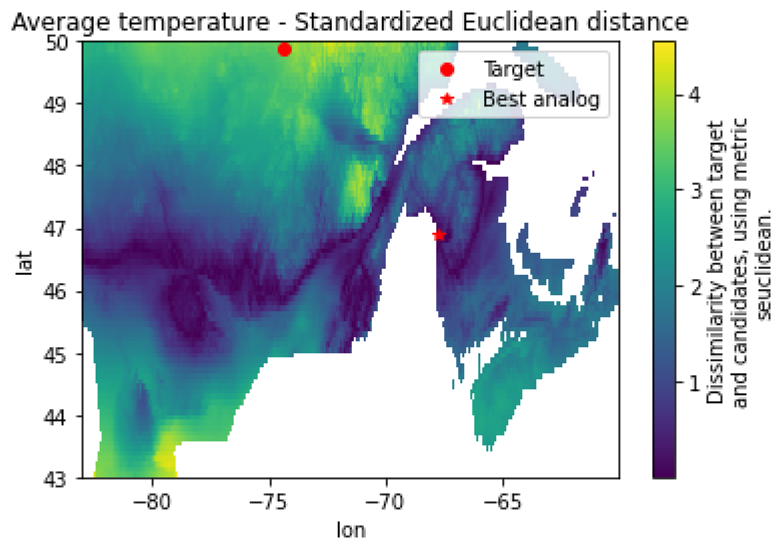
There are many metrics available to compute the dissimilarity between the indicator distributions. For our first test, we'll use the mean annual temperature (`tg_mean`) and the simple standardized euclidean distance metric (`seuclidean`). This is a very basic metric that only computes the distance between the means. All algorithms used to compare distributions are available through the `xclim.analog.spatial_analogs` function. They also live as well-documented functions in the same module or in the `xclim.analog.metrics` dictionary.

[6]:
```
results = analog.spatial_analogs(
    sim[["tg_mean"]], obs[["tg_mean"]], method="seuclidean"
)

results.plot()
plt.plot(sim.lon, sim.lat, "ro", label="Target")
```

(continues on next page)

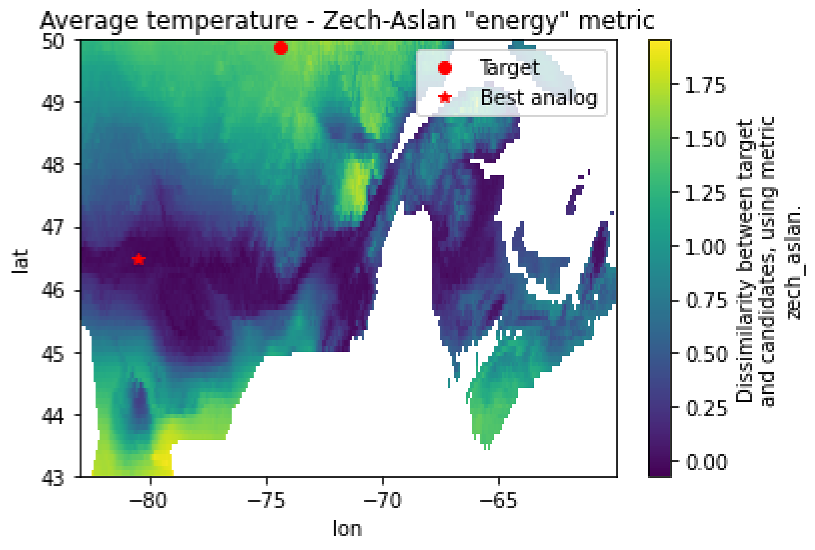
(continued from previous page)

```
def plot_best_analog(scores, ax=None):
    scores1d = scores.stack(site=["lon", "lat"])
    lon, lat = scores1d.isel(site=scores1d.argmin("site")).site.item()
    ax = ax or plt.gca()
    ax.plot(lon, lat, "r*", label="Best analog")


plot_best_analog(results)
plt.title("Average temperature - Standardized Euclidean distance")
plt.legend();
```


Average temperature - Standardized Euclidean distance
Target
Best analog
lat
lon
Dissimilarity between target and candidates, using metric seuclidean.

This shows that the average temperature projected by our simulation for Chibougamau in 2041-2070 will be similar to the 1981-2010 average temperature of a region approximately extending zonally between 46°N and 47°N. Evidently, this metric is limited as it only compares the time averages. Let's run this again with the "Zech-Aslan" metric, one that compares the whole distribution.

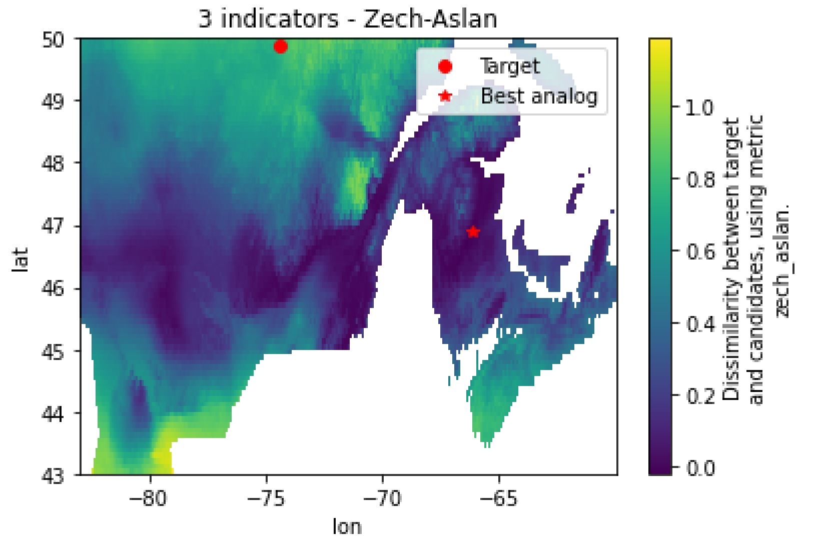
```
[7]: results = analog.spatial_analogs(
         sim[["tg_mean"]], obs[["tg_mean"]], method="zech_aslan"
     )

     results.plot(center=False)
     plt.plot(sim.lon, sim.lat, "ro", label="Target")
     plot_best_analog(results)
     plt.title('Average temperature - Zech-Aslan "energy" metric')
     plt.legend();
```


Average temperature - Zech-Aslan "energy" metric
Target
Best analog
lat
lon
Dissimilarity between target and candidates, using metric zech_aslan.

The new map is quite similar to the previous one, but notice how the scale has changed. Each metric defines its own scale (see the docstrings), but in all cases, lower values imply less differences between distributions. Notice also how the best analog has moved. This illustrates a common issue with these computations : there's a lot of noise in the results and the absolute minimum may be extremely sensitive and move all over the place.

These univariate analogies are interesting, but the real power of this method is that it can perform multivariate analyses.

[8]:
```python
results = analog.spatial_analogs(sim, obs, method="zech_aslan")

results.plot(center=False)
plt.plot(sim.lon, sim.lat, "ro", label="Target")
plot_best_analog(results)
plt.legend()
plt.title("3 indicators - Zech-Aslan");
```

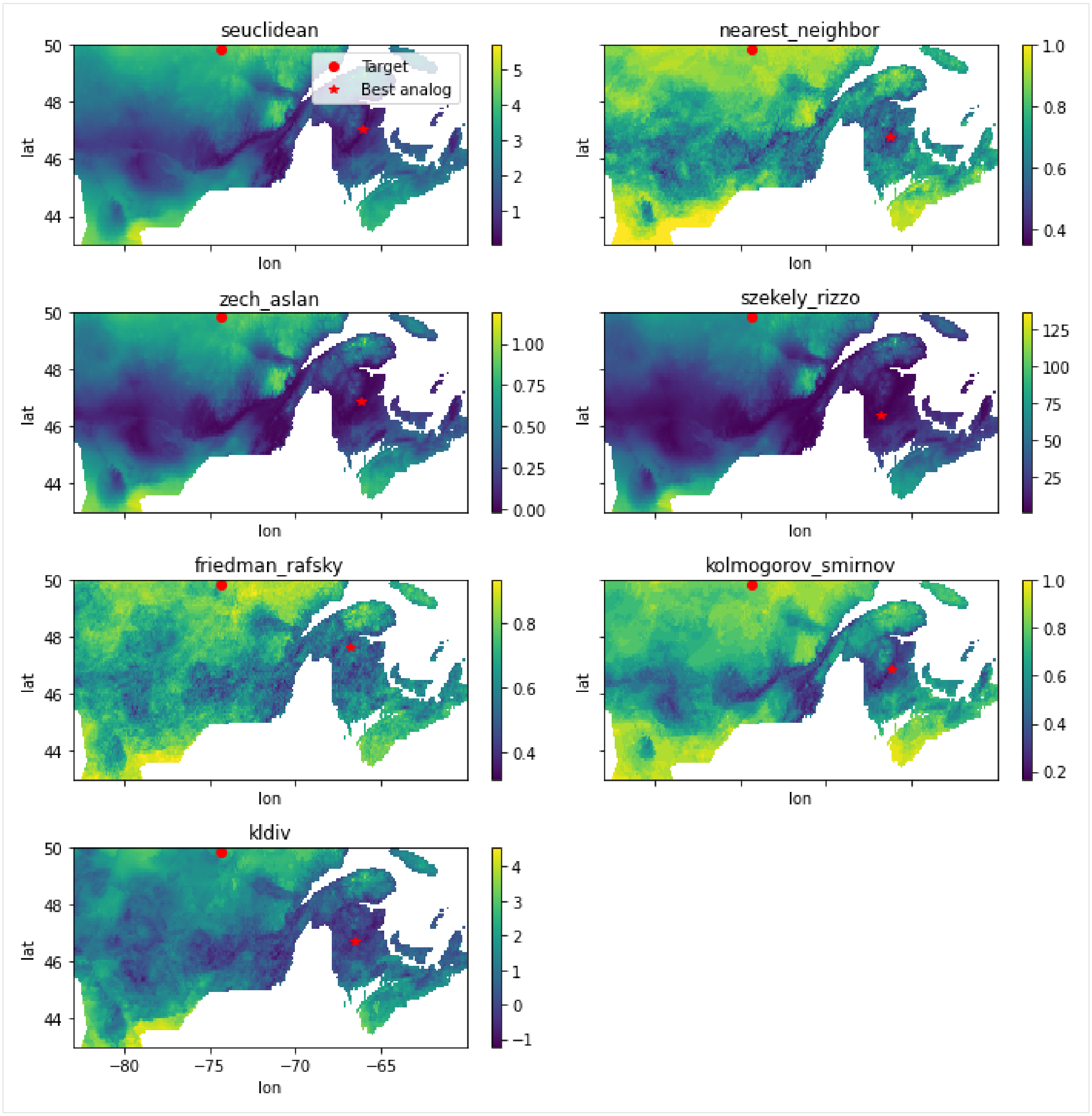

3 indicators - Zech-Aslan
Target
Best analog
lat
lon
Dissimilarity between target and candidates, using metric zech_aslan.

As said just above, results depend on the metric used. For example, some of the metrics include some sort of standardization while others don't. In the latter case, this means the absolute magnitude of the indices influences the results, i.e. analogies depend on the units. This information is written in the docstring.

Some are also much more efficient than other (for example : `seuclidean` or `zech_aslan`, compared to `kolmogorov_smirnov` or `friedman_rafsky`).

[9]:
```
# This cell is slow.
import time

fig, axs = plt.subplots(4, 2, sharex=True, sharey=True, figsize=(10, 10))
for metric, ax in zip(analog.metrics.keys(), axs.flatten()):
    start = time.perf_counter()
    results = analog.spatial_analogs(sim, obs, method=metric)
    print(f"Metric {metric} took {time.perf_counter() - start:.0f} s.")

    results.plot(center=False, ax=ax, cbar_kwargs={"label": ""})
    ax.plot(sim.lon, sim.lat, "ro", label="Target")
    plot_best_analog(results, ax=ax)
    ax.set_title(metric)
axs[0, 0].legend()
axs[-1, -1].set_visible(False)
fig.tight_layout();
```

```
Metric seuclidean took 2 s.
Metric nearest_neighbor took 8 s.
Metric zech_aslan took 4 s.
Metric szekely_rizzo took 3 s.
Metric friedman_rafsky took 21 s.
Metric kolmogorov_smirnov took 14 s.
Metric kldiv took 11 s.
```


seuclidean
nearest_neighbor
zech_aslan
szekely_rizzo
friedman_rafsky
kolmogorov_smirnov
kldiv
Target
Best analog
lat
lon
50
48
46
44
−80
−75
−70
−65

CHAPTER

FOUR

CLIMATE INDICATORS

xclim.core.indicator.Indicator instances essentially perform the same computations as the functions found in the *xclim.indices* library, but also run a number of health checks on input data and assign attributes to the output arrays. So for example, if there are missing values in a time series, indices won't notice, but indicators will return NaNs for periods with missing values (depending on the missing values algorithm selected, see *Missing values identification*). Indicators also check that the input data has the expected frequency (e.g. daily) and that it is indeed the expected variable (e.g. a precipitation flux). The output is assigned attributes that conform as much as possible with the CF-Convention.

Indicators are split into realms (atmos, land, seaIce), according to the variables they operate on. See *Defining new indicators* for instruction on how to create your own indicators. This page lists all indicators with a summary description, click on the names to get to the complete docstring of each indicator.

4.1 atmos: Atmosphere

4.2 land: Land surface

4.3 seaIce: Sea ice

4.4 Virtual submodules

4.4.1 CF Standard indices

Indicator found here are defined by the team at clix-meta. Adapted documentation from that repository follows:

The repository aims to provide a platform for thinking about, and developing, a unified view of metadata elements required to describe climate indices (aka climate indicators).

To facilitate data exchange and dissemination the metadata should, as far as possible, follow the Climate and Forecasting (CF) Conventions. Considering the very rich and diverse flora of climate indices this is however not always possible. By collecting a wide range of different indices it is easier to discover any common patterns and features that are currently not well covered by the CF Conventions. Currently identified issues frequently relate to standard_name or/and cell_methods which both are controlled vocabularies of the CF Conventions.

4.4.2 ICCLIM indices

The European Climate Assessment & Dataset project (ECAD) defines a set of 26 core climate indices. Those have been made accessible directly in xclim through their ECAD name for compatibility. However, the methods in this module are only wrappers around the corresponding methods of *xclim.indices*. Note that none of the checks performed by the *xclim.utils.Indicator* class (like with *xclim.atmos* indicators) are performed in this module.

4.4.3 ANUCLIM indices

The ANUCLIM (v6.1) software package BIOCLIM sub-module produces a set of bioclimatic parameters derived values of temperature and precipitation. The methods in this module are wrappers around a subset of corresponding methods of *xclim.indices*.

Furthermore, according to the ANUCLIM user-guide ([ANUCLIM]), input values should be at a weekly or monthly frequency. However, the implementation here expands these definitions and can calculate the result with daily input data.

CHAPTER

FIVE

HEALTH CHECKS

The `Indicator` class performs a number of sanity checks on inputs to make sure valid data is fed to indices computations (*`cfchecks`* for checks on the metadata and *`datachecks`* for checks on the coordinates). Output values are properly masked in case input values are missing or invalid (*`missing`*). Finally, a user can use functions of *`dataflags`* to explore potential issues with its data (extreme values, suspicious runs, etc).

5.1 CF-Convention checking

Utilities designed to verify the compliance of metadata with the CF-Convention.

`xclim.core.cfchecks.cfcheck_from_name(`*varname*, *vardata*`)`

Perform cfchecks on a DataArray using specifications from xclim's default variables.

`xclim.core.cfchecks.check_valid(`*var*, *key: str*, *expected: str | Sequence[str]*`)`

Check that a variable's attribute has one of the expected values. Raise a ValidationError otherwise.

5.2 Data checks

Utilities designed to check the validity of data inputs.

`xclim.core.datachecks.check_daily(`*var: DataArray*`)`

Raise an error if not series has a frequency other that daily, or is not monotonically increasing.

Note that this does not check for gaps in the series.

`xclim.core.datachecks.check_freq(`*var: xr.DataArray*, *freq: str | Sequence[str]*, *strict: bool = True*`)`

Raise an error if not series has not the expected temporal frequency or is not monotonically increasing.

Parameters

- **var** (*xr.DataArray*) – Input array.
- **freq** (*str or sequence of str*) – The expected temporal frequencies, using Pandas frequency terminology ({'A', 'M', 'D', 'H', 'T', 'S', 'L', 'U'} and multiples thereof). To test strictly for 'W', pass '7D' with *strict=True*. This ignores the start flag and the anchor (ex: 'AS-JUL' will validate against 'Y').
- **strict** (*bool*) – Whether multiples of the frequencies are considered invalid or not. With *strict* set to False, a '3H' series will not raise an error if freq is set to 'H'.

5.3 Missing values identification

Indicators may use different criteria to determine whether a computed indicator value should be considered missing. In some cases, the presence of any missing value in the input time series should result in a missing indicator value for that period. In other cases, a minimum number of valid values or a percentage of missing values should be enforced. The World Meteorological Organisation (WMO) suggests criteria based on the number of consecutive and overall missing values per month.

xclim has a registry of missing value detection algorithms that can be extended by users to customize the behavior of indicators. Once registered, algorithms can be used within indicators by setting the *missing* attribute of an *Indicator* subclass. By default, *xclim* registers the following algorithms:

- *any*: A result is missing if any input value is missing.
- *at_least_n*: A result is missing if less than a given number of valid values are present.
- *pct*: A result is missing if more than a given fraction of values are missing.
- *wmo*: A result is missing if 11 days are missing, or 5 consecutive values are missing in a month.
- *skip*: Skip missing value detection.
- *from_context*: Look-up the missing value algorithm from options settings. See `xclim.set_options()`.

To define another missing value algorithm, subclass `MissingBase` and decorate it with *`xclim.core.options.register_missing_method()`*.

Corresponding stand-alone functions are also exposed to run the same missing value checks independent from indicator calculations.

`xclim.core.missing.missing_any(`*da*, *freq*, *src_timestep=None*, ***indexer*`)`

Return whether there are missing days in the array.

Parameters

- **da** (*DataArray*) – Input array.
- **freq** (*str*) – Resampling frequency.
- **src_timestep** (*{"D", "H", "M"}*) – Expected input frequency.
- **indexer** (*{dim: indexer, }, optional*) – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered.

Returns

DataArray – A boolean array set to True if period has missing values.

`xclim.core.missing.at_least_n_valid(`*da*, *freq*, *n=1*, *src_timestep=None*, ***indexer*`)`

Return whether there are at least a given number of valid values.

Parameters

- **da** (*DataArray*) – Input array.
- **freq** (*str*) – Resampling frequency.
- **n** (*int*) – Minimum of valid values required.
- **src_timestep** (*{"D", "H"}*) – Expected input frequency.

- **indexer** (*{dim: indexer, }, optional*) – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered.

Returns
out (*DataArray*) – A boolean array set to True if period has missing values.

`xclim.core.missing.missing_pct(`*da*, *freq*, *tolerance*, *src_timestep=None*, ***indexer*`)`

Return whether there are more missing days in the array than a given percentage.

Parameters

- **da** (*DataArray*) – Input array.
- **freq** (*str*) – Resampling frequency.
- **tolerance** (*float*) – Fraction of missing values that are tolerated [0,1].
- **src_timestep** (*{"D", "H"}*) – Expected input frequency.
- **indexer** (*{dim: indexer, }, optional*) – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered.

Returns
DataArray – A boolean array set to True if period has missing values.

`xclim.core.missing.missing_wmo(`*da*, *freq*, *nm=11*, *nc=5*, *src_timestep=None*, ***indexer*`)`

Return whether a series fails WMO criteria for missing days.

The World Meteorological Organisation recommends that where monthly means are computed from daily values, it should be considered missing if either of these two criteria are met:

– observations are missing for 11 or more days during the month; – observations are missing for a period of 5 or more consecutive days during the month.

Stricter criteria are sometimes used in practice, with a tolerance of 5 missing values or 3 consecutive missing values.

Parameters

- **da** (*DataArray*) – Input array.
- **freq** (*str*) – Resampling frequency.
- **nm** (*int*) – Number of missing values per month that should not be exceeded.
- **nc** (*int*) – Number of consecutive missing values per month that should not be exceeded.
- **src_timestep** (*{"D"}*) – Expected input frequency. Only daily values are supported.
- **indexer** (*{dim: indexer, }, optional*) – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered.

Returns
DataArray – A boolean array set to True if period has missing values.

Notes

If used at frequencies larger than a month, for example on an annual or seasonal basis, the function will return True if any month within a period is missing.

`xclim.core.missing.missing_from_context(`*da*, *freq*, *src_timestep=None*, ***indexer*`)`

Return whether each element of the resampled da should be considered missing according to the currently set options in *xclim.set_options*.

See *xclim.set_options* and *xclim.core.options.register_missing_method*.

5.4 Data flags

Pseudo-indicators designed to analyse supplied variables for suspicious/erroneous indicator values.

`exception xclim.core.dataflags.DataQualityException(`*flag_array: Dataset*, *message='Data quality flags indicate suspicious values. Flags raised are:\n - '*`)`

Bases: `Exception`

Raised when any data evaluation checks are flagged as True.

Variables

- `flag_array` (*xarray.Dataset*) – Xarray.Dataset of Data Flags.
- `message` (*str*) – Message prepended to the error messages.

`xclim.core.dataflags.data_flags(`*da: xarray.DataArray*, *ds: xarray.Dataset | None = None*, *flags: dict | None = None*, *dims: None | str | Sequence[str] = 'all'*, *freq: str | None = None*, *raise_flags: bool = False*`)` → xarray.Dataset

Evaluate the supplied DataArray for a set of data flag checks.

Test triggers depend on variable name and availability of extra variables within Dataset for comparison. If called with *raise_flags=True*, will raise a DataQualityException with comments for each failed quality check.

Parameters

- **da** (*xarray.DataArray*) – The variable to check. Must have a name that is a valid CMIP6 variable name and appears in `xclim.core.utils.VARIABLES`.
- **ds** (*xarray.Dataset, optional*) – An optional dataset with extra variables needed by some checks.
- **flags** (*dict, optional*) – A dictionary where the keys are the name of the flags to check and the values are parameter dictionaries. The value can be None if there are no parameters to pass (i.e. default will be used). The default, None, means that the data flags list will be taken from `xclim.core.utils.VARIABLES`.
- **dims** (*{"all", None} or str or a sequence of strings*) – Dimenions upon which aggregation should be performed. Default: "all".
- **freq** (*str, optional*) – Resampling frequency to have data_flags aggregated over periods. Defaults to None, which means the "time" axis is treated as any other dimension (see *dims*).
- **raise_flags** (*bool*) – Raise exception if any of the quality assessment flags are raised. Default: False.

Returns
xarray.Dataset

Examples

To evaluate all applicable data flags for a given variable:

```
>>> from xclim.core.dataflags import data_flags
>>> ds = xr.open_dataset(path_to_pr_file)
>>> flagged = data_flags(ds.pr, ds)
```

The next example evaluates only one data flag, passing specific parameters. It also aggregates the flags yearly over the "time" dimension only, such that a True means there is a bad data point for that year at that location.

```
>>> flagged = data_flags(
...     ds.pr,
...     ds,
...     flags={"very_large_precipitation_events": {"thresh": "250 mm d-1"}},
...     dims=None,
...     freq="YS",
... )
```

`xclim.core.dataflags.ecad_compliant(`*ds: xarray.Dataset*, *dims: None | str | Sequence[str] = 'all'*, *raise_flags: bool = False*, *append: bool = True*`)` → xarray.DataArray | xarray.Dataset | None

Run ECAD compliance tests.

Assert file adheres to ECAD-based quality assurance checks.

Parameters

- **ds** (*xarray.Dataset*) – Dataset containing variables to be examined.
- **dims** (*{"all", None} or str or a sequence of strings*) – Dimensions upon which aggregation should be performed. Default: "all".
- **raise_flags** (*bool*) – Raise exception if any of the quality assessment flags are raised, otherwise returns None. Default: False.
- **append** (*bool*) – If *True*, returns the Dataset with the *ecad_qc_flag* array appended to data_vars. If *False*, returns the DataArray of the *ecad_qc_flag* variable.

Returns
Union[xarray.DataArray, xarray.Dataset]

`xclim.core.dataflags.negative_accumulation_values(`*da: DataArray*`)` → DataArray

Check if variable values are negative for any given day.

Parameters
da (*xarray.DataArray*)

Returns
xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import negative_accumulation_values
>>> ds = xr.open_dataset(path_to_pr_file)
>>> flagged = negative_accumulation_values(ds.pr)
```

`xclim.core.dataflags.outside_n_standard_deviations_of_climatology`(*da: DataArray*, ***, *n: int*, *window: int = 5*) → DataArray

Check if any daily value is outside n standard deviations from the day of year mean.

Parameters

- **da** (*xarray.DataArray*) – The DataArray being examined.
- **n** (*int*) – Number of standard deviations.
- **window** (*int*) – Moving window used to determining climatological mean. Default: 5.

Returns

xarray.DataArray, [bool]

Notes

A moving window of 5 days is suggested for tas data flag calculations according to ICCLIM data quality standards.

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import outside_n_standard_deviations_of_climatology
>>> ds = xr.open_dataset(path_to_tas_file)
>>> std_devs = 5
>>> average_over = 5
>>> flagged = outside_n_standard_deviations_of_climatology(
...     ds.tas, n=std_devs, window=average_over
... )
```

`xclim.core.dataflags.percentage_values_outside_of_bounds`(*da: DataArray*) → DataArray

Check if variable values fall below 0% or rise above 100% for any given day.

Parameters

da (*xarray.DataArray*)

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array: >>> from xclim.core.dataflags import percentage_values_outside_of_bounds >>> ds = xr.open_dataset(path_to_huss_file) # doctest: +SKIP >>> flagged = percentage_values_outside_of_bounds(ds.huss) # doctest: +SKIP

`xclim.core.dataflags.register_methods`(*func*)

Summarize all methods used in dataflags checks.

`xclim.core.dataflags.tas_below_tasmin`(*tas: DataArray*, *tasmin: DataArray*) → DataArray

Check if tas values are below tasmin values for any given day.

Parameters

- **tas** (*xarray.DataArray*)
- **tasmin** (*xarray.DataArray*)

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import tas_below_tasmin
>>> ds = xr.open_dataset(path_to_tas_file)
>>> flagged = tas_below_tasmin(ds.tas, ds.tasmin)
```

`xclim.core.dataflags.tas_exceeds_tasmax`(*tas: DataArray*, *tasmax: DataArray*) → DataArray

Check if tas values tasmax values for any given day.

Parameters

- **tas** (*xarray.DataArray*)
- **tasmax** (*xarray.DataArray*)

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import tas_exceeds_tasmax
>>> ds = xr.open_dataset(path_to_tas_file)
>>> flagged = tas_exceeds_tasmax(ds.tas, ds.tasmax)
```

`xclim.core.dataflags.tasmax_below_tasmin`(*tasmax: DataArray*, *tasmin: DataArray*) → DataArray

Check if tasmax values are below tasmin values for any given day.

Parameters

- **tasmax** (*xarray.DataArray*)
- **tasmin** (*xarray.DataArray*)

Returns
xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import tasmax_below_tasmin
>>> ds = xr.open_dataset(path_to_tas_file)
>>> flagged = tasmax_below_tasmin(ds.tasmax, ds.tasmin)
```

`xclim.core.dataflags.temperature_extremely_high(`*da: DataArray*, ***, *thresh: str = '60 degC'*`)` → DataArray

Check if temperatures values exceed 60 degrees Celsius for any given day.

Parameters

- **da** (*xarray.DataArray*)
- **thresh** (*str*)

Returns
xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import temperature_extremely_high
>>> ds = xr.open_dataset(path_to_tas_file)
>>> temperature = "60 degC"
>>> flagged = temperature_extremely_high(ds.tas, thresh=temperature)
```

`xclim.core.dataflags.temperature_extremely_low(`*da: DataArray*, ***, *thresh: str = '-90 degC'*`)` → DataArray

Check if temperatures values are below -90 degrees Celsius for any given day.

Parameters

- **da** (*xarray.DataArray*)
- **thresh** (*str*)

Returns
xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import temperature_extremely_low
>>> ds = xr.open_dataset(path_to_tas_file)
>>> temperature = "-90 degC"
>>> flagged = temperature_extremely_low(ds.tas, thresh=temperature)
```

`xclim.core.dataflags.values_op_thresh_repeating_for_n_or_more_days(`*da: DataArray*, **, *n: int*, *thresh: str*, *op: str = 'eq'*`)` → DataArray

Check if array values repeat at a given threshold for 'n' or more days.

Parameters

- **da** (*xarray.DataArray*) – The DataArray being examined.
- **n** (*int*) – Number of days needed to trigger flag.
- **thresh** (*str*) – Repeating values to search for that will trigger flag.
- **op** (*{"eq", "gt", "lt", "gteq", "lteq"}*) – Operator used for comparison with thresh.

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import values_op_thresh_repeating_for_n_or_more_days
>>> ds = xr.open_dataset(path_to_pr_file)
>>> units = "5 mm d-1"
>>> days = 5
>>> comparison = "eq"
>>> flagged = values_op_thresh_repeating_for_n_or_more_days(
...     ds.pr, n=days, thresh=units, op=comparison
... )
```

`xclim.core.dataflags.values_repeating_for_n_or_more_days(`*da: DataArray*, **, *n: int*`)` → DataArray

Check if exact values are found to be repeating for at least 5 or more days.

Parameters

- **da** (*xarray.DataArray*) – The DataArray being examined.
- **n** (*int*) – Number of days to trigger flag.

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import values_repeating_for_n_or_more_days
>>> ds = xr.open_dataset(path_to_pr_file)
>>> flagged = values_repeating_for_n_or_more_days(ds.pr, n=5)
```

`xclim.core.dataflags.very_large_precipitation_events(`*da: DataArray, *, thresh='300 mm d-1'*`)` → DataArray

Check if precipitation values exceed 300 mm/day for any given day.

Parameters

- **da** (*xarray.DataArray*) – The DataArray being examined.
- **thresh** (*str*) – Threshold to search array for that will trigger flag if any day exceeds value.

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import very_large_precipitation_events
>>> ds = xr.open_dataset(path_to_pr_file)
>>> rate = "300 mm d-1"
>>> flagged = very_large_precipitation_events(ds.pr, thresh=rate)
```

`xclim.core.dataflags.wind_values_outside_of_bounds(`*da: DataArray, *, lower: str = '0 m s-1', upper: str = '46 m s-1'*`)` → DataArray

Check if variable values fall below 0% or rise above 100% for any given day.

Parameters

- **da** (*xarray.DataArray*) – The DataArray being examined.
- **lower** (*str*) – The lower limit for wind speed.
- **upper** (*str*) – The upper limit for wind speed.

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array: >>> from xclim.core.dataflags import wind_values_outside_of_bounds >>> ds = xr.open_dataset(path_to_tas_file) >>> ceiling, floor = "46 m s-1", "0 m s-1" >>> flagged = wind_values_outside_of_bounds(ds.wsgsmax, upper=ceiling, lower=floor)

CHAPTER

SIX

UNIT HANDLING

[1]:
```
from __future__ import annotations

import xarray as xr

import xclim as xc

# Set display to HTML style (optional)
xr.set_options(display_style="html", display_width=50)

# import plotting stuff
import matplotlib.pyplot as plt

%matplotlib inline
plt.style.use("seaborn")
plt.rcParams["figure.figsize"] = (11, 5)
```

A lot of effort has been placed into automatic handling of input data units. `xclim` will automatically detect the input variable(s) units (e.g. °C versus °K or mm/s versus mm/day etc.) and adjust on-the-fly in order to calculate indices in the consistent manner. This comes with the obvious caveat that input data requires metadata attribute for units

For precipitation data, `xclim` expects precipitation fluxes. This could be units of `length/time`, such as mm/d, or units of `mass / area / time`, such as $kg/m^2/s$. Units of `length` only, such as mm, are not supported, because the interpretation depends on the frequency of the data, which cannot always be inferred explicitly from the data. For example, if a daily precipitation series records total daily precipitation and has units of `mm`, change the units attribute to `mm/d` before computing indicators. Note that `xclim` will automatically convert between `mass / area / time` and `length/time` using a water density of 1000 kg/m^3 when the context is hydrology.

In the following examples, our toy temperature dataset comes in units of Kelvins (`"degK"`).

[2]:
```
# See the Usage page for details on opening datasets, subsetting and resampling.
ds = xr.tutorial.open_dataset("air_temperature")
tas = (
    ds.air.sel(lat=40, lon=270, method="nearest")
    .resample(time="D")
    .mean(keep_attrs=True)
)
print(tas.attrs["units"])
```

```
degK
```

Using pint, `xclim` provides useful functions to convert the units of datasets and `DataArrays`. Here, we convert our kelvin data to the very useful Fahrenheits:

[3]:
```
tas_F = xc.units.convert_units_to(tas, "degF")
print(tas_F.attrs["units"])
```

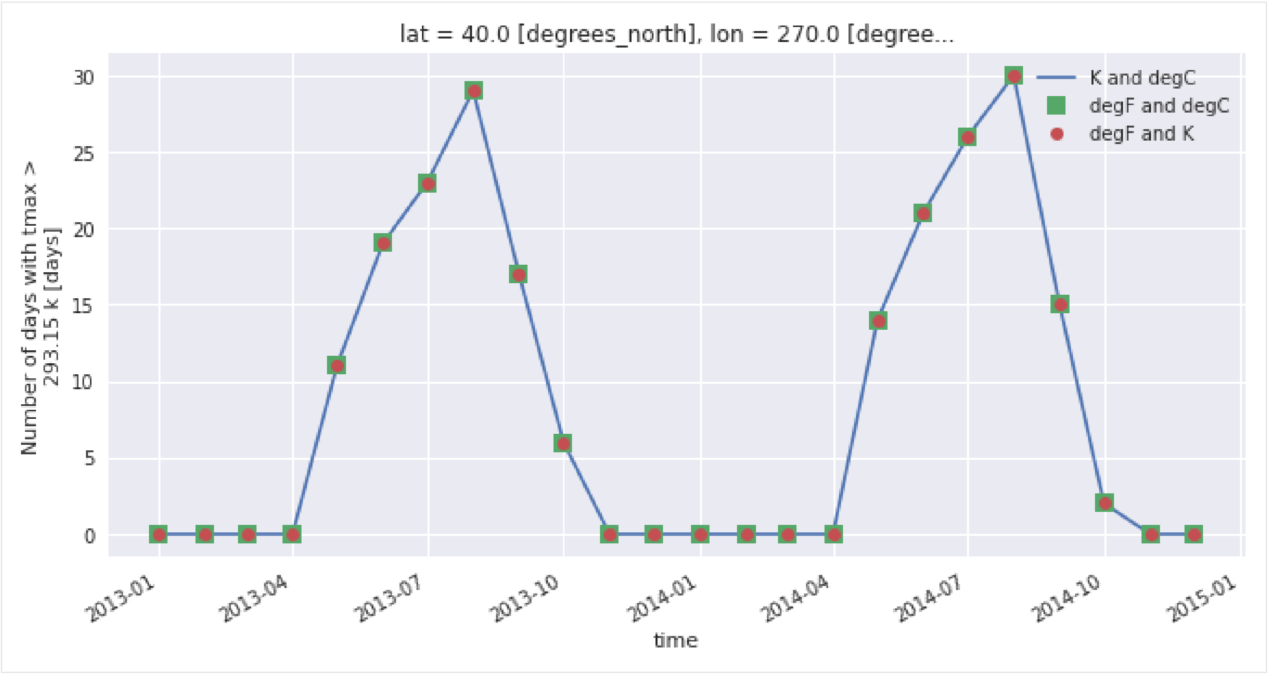
```
°F
```

6.1 Threshold indices

`xclim` unit handling also applies to threshold indicators. Users can provide threshold in units of choice and `xclim` will adjust automatically. For example determining the number of days with tasmax $>$ 20°C users can define a threshold input of '20 C' or '20 degC' even if input data is in Kelvin. Alernatively users can even provide a threshold in Kelvin '293.15 K' (if they really wanted to).

[4]:
```
with xc.set_options(cf_compliance="log"):
    # Using Kelvin data, threshold in Celsius
    out1 = xc.atmos.tx_days_above(tasmax=tas, thresh="20 C", freq="MS")

    # Using Fahrenheit data, threshold in Celsius
    out2 = xc.atmos.tx_days_above(tasmax=tas_F, thresh="20 C", freq="MS")

    # Using Fahrenheit data, with threshold in Kelvin
    out3 = xc.atmos.tx_days_above(tasmax=tas_F, thresh="293.15 K", freq="MS")

# Plot and see that it's all identical:
plt.figure()
out1.plot(label="K and degC", linestyle="-")
out2.plot(label="degF and degC", marker="s", markersize=10, linestyle="none")
out3.plot(label="degF and K", marker="o", linestyle="none")
plt.legend()
```

[4]:
```
<matplotlib.legend.Legend at 0x7f6d3c883b20>
```



6.2 Sum and count indices

Many indices in `xclim` will either sum values or count events along the time dimension and over a period. As of version 0.24, unit handling dynamically infers what the sampling frequency and its corresponding unit is.

Indicators, on the other hand, do not have this flexibility and often **expect** input at a given frequency, more often daily then otherwise.

For example, we can run the `tx_days_above` on the 6-hourly test data and it should return similar results as on the daily data, but in units of `h` (the base unit of the sampling frequency).

[5]:
```
tas_6h = ds.air.sel(
    lat=40, lon=270, method="nearest"
)  # no resampling, original data is 6-hourly
out4_h = xc.indices.tx_days_above(tasmax=tas_6h, thresh="20 C", freq="MS")
out4_h
```

[5]:
```
<xarray.DataArray 'air' (time: 24)>
array([  0,    0,    0,   48, 228, 426, 492, 612, 456, 174,    0,    0,    0,
         0,    0,   54, 282, 552, 504, 636, 324,   78,    0,    0])
Coordinates:
  * time     (time) datetime64[ns] 2013-01-01 ...
    lat      float32 40.0
    lon      float32 270.0
Attributes:
    units:    h
```

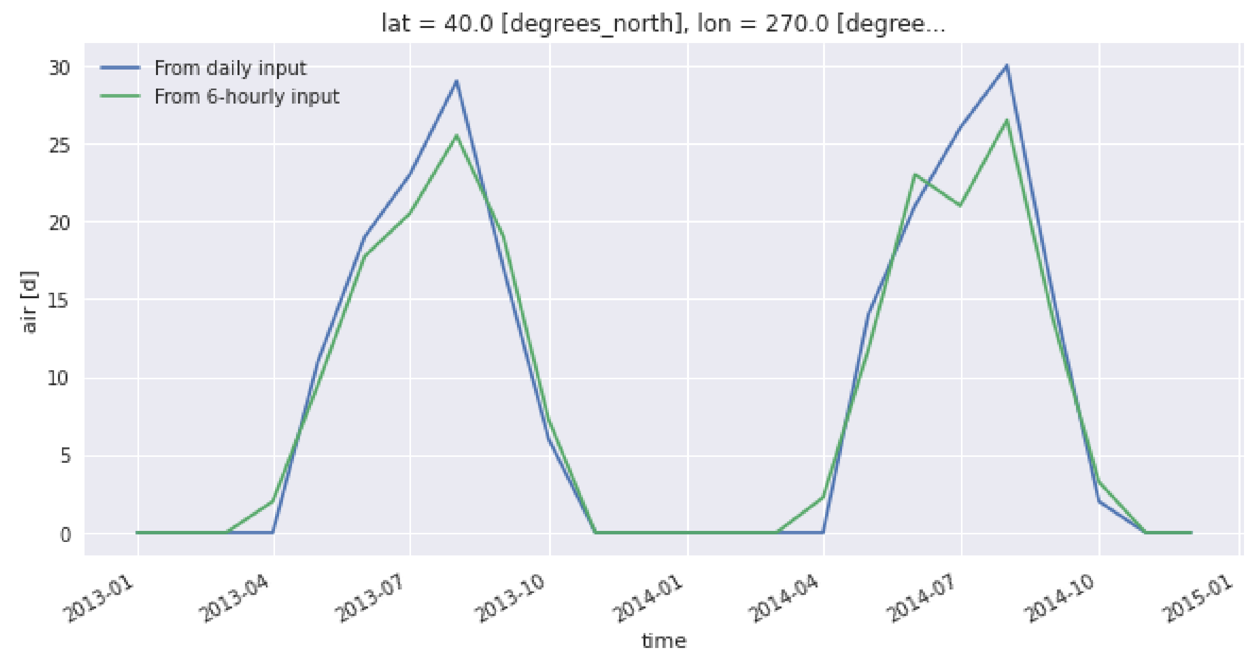
[6]:
```
out4_d = xc.units.convert_units_to(out4_h, "d")
plt.figure()
```

(continues on next page)

(continued from previous page)

```
out1.plot(label="From daily input", linestyle="-")
out4_d.plot(label="From 6-hourly input", linestyle="-")
plt.legend()
```

[6]:
```
<matplotlib.legend.Legend at 0x7f6d3c7fc310>
```



6.2.1 Other utilites

Many helper functions are defined in `xclim.core.units`, see *Unit handling module*.

CHAPTER

SEVEN

INTERNATIONALIZATION

This module defines methods and object to help the internationalization of metadata for climate indicators computed by xclim. Go to *Adding translated metadata* to see how to use this feature.

All the methods and objects in this module use localization data given in json files. These files are expected to be defined as in this example for french:

```
{
    "attrs_mapping": {
        "modifiers": ["", "f", "mpl", "fpl"],
        "YS": ["annuel", "annuelle", "annuels", "annuelles"],
        "AS-*": ["annuel", "annuelle", "annuels", "annuelles"],
        # ... and so on for other frequent parameters translation...
    },
    "DTRVAR": {
        "long_name": "Variabilité de l'amplitude de la température diurne",
        "description": "Variabilité {freq:f} de l'amplitude de la température diurne␣
↪(définie comme la moyenne de la variation journalière de l'amplitude de température␣
↪sur une période donnée)",
        "title": "Variation quotidienne absolue moyenne de l'amplitude de la température␣
↪diurne",
        "comment": "",
        "abstract": "La valeur absolue de la moyenne de l'amplitude de la température␣
↪diurne.",
    },
    # ... and so on for other indicators...
}
```

Indicators are named by subclass identifier, the same as in the indicator registry (*xclim.core.indicators.registry*), but which can differ from the callable name. In this case, the indicator is called through *atmos.daily_temperature_range_variability*, but its identifier is *DTRVAR*. Use the *ind.__class__.__name__* accessor to get its registry name.

Here, the usual parameter passed to the formatting of "description" is "freq" and is usually translated from "YS" to "annual". However, in french and in this sentence, the feminine form should be used, so the "f" modifier is added by the translator so that the formatting function knows which translation to use. Acceptable entries for the mappings are limited to what is already defined in *xclim.core.indicators.utils.default_formatter*.

For user-provided internationalization dictionaries, only the "attrs_mapping" and its "modifiers" key are mandatory, all other entries (translations of frequent parameters and all indicator entries) are optional. For xclim-provided translations (for now only french), all indicators must have en entry and the "attrs_mapping" entries must match exactly the default formatter. Those default translations are found in the *xclim/locales* folder.

`xclim.core.locales.TRANSLATABLE_ATTRS = ['long_name', 'description', 'comment', 'title', 'abstract', 'keywords']`

List of attributes to consider translatable when generating locale dictionaries.

`exception xclim.core.locales.UnavailableLocaleError(`*locale*`)`

Bases: `ValueError`

Error raised when a locale is requested but doesn't exist.

`xclim.core.locales.generate_local_dict(`*locale: str*, *init_english: bool = False*`)` → dict

Generate a dictionary with keys for each indicator and translatable attributes.

Parameters

- **locale** (*str*) – Locale in the IETF format
- **init_english** (*bool*) – If True, fills the initial dictionary with the english versions of the attributes. Defaults to False.

`xclim.core.locales.get_local_attrs(`*indicator: str | Sequence[str]*, **locales: str | Sequence[str] | tuple[str, dict]*, *names: Sequence[str] | None = None*, *append_locale_name: bool = True*`)` → dict

Get all attributes of an indicator in the requested locales.

Parameters

- **indicator** (*str or sequence of strings*) – Indicator's class name, usually the same as in *xc.core.indicator.registry*. If multiple names are passed, the attrs from each indicator are merged, with the highest priority set to the first name.
- **locales** (*str or tuple of str*) – IETF language tag or a tuple of the language tag and a translation dict, or a tuple of the language tag and a path to a json file defining translation of attributes.
- **names** (*Optional[Sequence[str]]*) – If given, only returns translations of attributes in this list.
- **append_locale_name** (*bool*) – If True (default), append the language tag (as "{attr_name}_{locale}") to the returned attributes.

Raises

`ValueError` – If *append_locale_name* is False and multiple *locales* are requested.

Returns

dict – All CF attributes available for given indicator and locales. Warns and returns an empty dict if none were available.

`xclim.core.locales.get_local_dict(`*locale: str | Sequence[str] | tuple[str, dict]*`)` → tuple[str, dict]

Return all translated metadata for a given locale.

Parameters

locale (*str or sequence of str*) – IETF language tag or a tuple of the language tag and a translation dict, or a tuple of the language tag and a path to a json file defining translation of attributes.

Raises

`UnavailableLocaleError` – If the given locale is not available.

Returns

- *str* – The best fitting locale string
- *dict* – The available translations in this locale.

`xclim.core.locales.get_local_formatter(`*locale: str | Sequence[str] | tuple[str, dict]*`)` → *AttrFormatter*

Return an AttrFormatter instance for the given locale.

Parameters

locale (*str or tuple of str*) – IETF language tag or a tuple of the language tag and a translation dict, or a tuple of the language tag and a path to a json file defining translation of attributes.

`xclim.core.locales.list_locales()`

List of loaded locales. Includes all loaded locales, no matter how complete the translations are.

`xclim.core.locales.load_locale(`*locdata: str | Path | Mapping[str, dict]*, *locale: str*`)`

Load translations from a json file into xclim.

Parameters

- **locdata** (*str or dictionary*) – Either a loaded locale dictionary or a path to a json file.
- **locale** (*str*) – The locale name (IETF tag).

`xclim.core.locales.read_locale_file(`*filename*, *module: str | None = None*, *encoding: str = 'UTF8'*`)` → dict

Read a locale file (.json) and return its dictionary.

Parameters

- **filename** (*PathLike*) – The file to read.
- **module** (*str, optional*) – If module is a string, this module name is added to all identifiers translated in this file. Defaults to None, and no module name is added (as if the indicator was an official xclim indicator).
- **encoding** (*str*) – The encoding to use when reading the file. Defaults to UTF-8, overriding python's default mechanism which is machine dependent.

CHAPTER

EIGHT

COMMAND LINE INTERFACE

xclim provides the `xclim` command line executable to perform basic indicator computation easily without having to start up a full python environment. However, not all indicators listed in *Climate Indicators* are available through this tool.

Its use is simple. Type the following to see the usage message:

[1]:
```
!xclim --help
```

```
Usage: xclim [OPTIONS] INDICATOR1 [OPTIONS] ... [INDICATOR2 [OPTIONS] ... ]
             ...

  Command line tool to compute indices on netCDF datasets. Indicators are
  referred to by their (case-insensitive) identifier, as in
  xclim.core.indicator.registry.

Options:
  -i, --input TEXT         Input files. Can be a netCDF path or a glob
                           pattern.
  -o, --output TEXT        Output filepath. A new file will be created
  -v, --verbose            Print details about context and progress.
  -V, --version            Prints xclim's version number and exits
  --dask-nthreads INTEGER  Start a dask.distributed Client with this many
                           threads and 1 worker. If not specified, the local
                           scheduler is used. If specified, '--dask-maxmem'
                           must also be given
  --dask-maxmem TEXT       Memory limit for the dask.distributed Client as a
                           human readable string (ex: 4GB). If specified, '--
                           dask-nthreads' must also be specified.
  --chunks TEXT            Chunks to use when opening the input dataset(s).
                           Given as <dim1>:num,<dim2:num>. Ex:
                           time:365,lat:168,lon:150.
  --help                   Show this message and exit.

Commands:
  indices            List indicators.
  info               Give information about INDICATOR.
  dataflags          Run data flag checks for input variables.
  release_notes      Print history for publishing purposes.
  show_version_info  Print versions of dependencies for debugging purposes.
```

To list all available indicators, use the "indices" subcommand:

[2]:
```
!xclim indices
```

```
Listing all available indicators for computation.:
anuclim.p10_meantempwarmestquarter
                                P10_MeanTempWarmestQuarter
                                (P10_MeanTempWarmestQuarter)
anuclim.p11_meantempcoldestquarter
                                P11_MeanTempColdestQuarter
                                (P11_MeanTempColdestQuarter)
anuclim.p12_annualprecip        Annual Precipitation (P12_AnnualPrecip)
anuclim.p13_precipwettestperiod
                                P13_PrecipWettestPeriod
                                (P13_PrecipWettestPeriod)
anuclim.p14_precipdriestperiod  P14_PrecipDriestPeriod
                                (P14_PrecipDriestPeriod)
anuclim.p15_precipseasonality   P15_PrecipSeasonality
                                (P15_PrecipSeasonality)
anuclim.p16_precipwettestquarter
                                P16_PrecipWettestQuarter
                                (P16_PrecipWettestQuarter)
anuclim.p17_precipdriestquarter
                                P17_PrecipDriestQuarter
                                (P17_PrecipDriestQuarter)
anuclim.p18_precipwarmestquarter
                                P18_PrecipWarmestQuarter
                                (P18_PrecipWarmestQuarter)
anuclim.p19_precipcoldestquarter
                                P19_PrecipColdestQuarter
                                (P19_PrecipColdestQuarter)
anuclim.p1_annmeantemp          Annual Mean Temperature (P1_AnnMeanTemp)
anuclim.p2_meandiurnalrange     Mean Diurnal Range (P2_MeanDiurnalRange)
anuclim.p3_isothermality        P3_Isothermality (P3_Isothermality)
anuclim.p4_tempseasonality      P4_TempSeasonality (P4_TempSeasonality)
anuclim.p5_maxtempwarmestperiod
                                Max Temperature of Warmest Period
                                (P5_MaxTempWarmestPeriod)
anuclim.p6_mintempcoldestperiod
                                Min Temperature of Coldest Period
                                (P6_MinTempColdestPeriod)
anuclim.p7_tempannualrange      Temperature Annual Range
                                (P7_TempAnnualRange)
anuclim.p8_meantempwettestquarter
                                P8_MeanTempWettestQuarter
                                (P8_MeanTempWettestQuarter)
anuclim.p9_meantempdriestquarter
                                P9_MeanTempDriestQuarter
                                (P9_MeanTempDriestQuarter)
base_flow_index                 Base flow index
biologically_effective_degree_days
                                Biologically effective degree days computed
                                with {method} formula (Summation of
                                min((max((Tmin + Tmax)/2 - {thresh_tasmin},
                                0) * k) + TR_adg, 9°C), for days between
```

(continues on next page)

(continued from previous page)

	{start_date} and {end_date}). (bedd)
blowing_snow	Number of days where snowfall and wind speeds are above respective thresholds. ({freq}_blowing_snow)
calm_days	Number of days with surface wind speed below threshold
cdd	Maximum consecutive dry days (Precip < {thresh})
cf.cdd	Maximum consecutive dry days (Precip < 1mm) (cdd)
cf.cddcoldtt	Cooling Degree Days (Tmean > {threshold}C) (cddcold{threshold})
cf.cfd	Maximum number of consecutive frost days (Tmin < 0 C) (cfd)
cf.csu	Maximum number of consecutive summer days (Tmax >25 C) (csu)
cf.ctmgett	Maximum number of consequtive days with Tmean >= {threshold}C (ctmge{threshold})
cf.ctmgttt	Maximum number of consequtive days with Tmean > {threshold}C (ctmgt{threshold})
cf.ctmlett	Maximum number of consequtive days with Tmean <= {threshold}C (ctmle{threshold})
cf.ctmlttt	Maximum number of consequtive days with Tmean < {threshold}C (ctmlt{threshold})
cf.ctngett	Maximum number of consequtive days with Tmin >= {threshold}C (ctnge{threshold})
cf.ctngttt	Maximum number of consequtive days with Tmin > {threshold}C (ctngt{threshold})
cf.ctnlett	Maximum number of consequtive days with Tmin <= {threshold}C (ctnle{threshold})
cf.ctnlttt	Maximum number of consequtive days with Tmin < {threshold}C (ctnlt{threshold})
cf.ctxgett	Maximum number of consequtive days with Tmax >= {threshold}C (ctxge{threshold})
cf.ctxgttt	Maximum number of consequtive days with Tmax > {threshold}C (ctxgt{threshold})
cf.ctxlett	Maximum number of consequtive days with Tmax <= {threshold}C (ctxle{threshold})
cf.ctxlttt	Maximum number of consequtive days with Tmax < {threshold}C (ctxlt{threshold})
cf.cwd	Maximum consecutive wet days (Precip >= 1mm) (cwd)
cf.ddgttt	Degree Days (Tmean > {threshold}C) (ddgt{threshold})
cf.ddlttt	Degree Days (Tmean < {threshold}C) (ddlt{threshold})
cf.dtr	Mean Diurnal Temperature Range (dtr)
cf.etr	Intra-period extreme temperature range (etr)
cf.fg	Mean of daily mean wind strength (fg)
cf.fxx	Maximum value of daily maximum wind gust strength (fxx)
cf.gd4	Growing degree days (sum of Tmean > 4 C)

(continues on next page)

(continued from previous page)

	(gd4)
cf.gddgrowtt	Annual Growing Degree Days (Tmean > {threshold}C) (gddgrow{threshold})
cf.hd17	Heating degree days (sum of Tmean < 17 C) (hd17)
cf.hddheattt	Heating Degree Days (Tmean < {threshold}C) (hddheat{threshold})
cf.maxdtr	Maximum Diurnal Temperature Range (maxdtr)
cf.pp	Mean of daily sea level pressure (pp)
cf.rh	Mean of daily relative humidity (rh)
cf.sd	Mean of daily snow depth (sd)
cf.sdii	Average precipitation during Wet Days (SDII) (sdii)
cf.ss	Sunshine duration, sum (ss)
cf.tg	Mean of daily mean temperature (tg)
cf.tmm	Mean daily mean temperature (tmm)
cf.tmmax	Maximum daily mean temperature (tmmax)
cf.tmmean	Mean daily mean temperature (tmmean)
cf.tmmin	Minimum daily mean temperature (tmmin)
cf.tmn	Minimum daily mean temperature (tmn)
cf.tmx	Maximum daily mean temperature (tmx)
cf.tn	Mean of daily minimum temperature (tn)
cf.tnm	Mean daily minimum temperature (tnm)
cf.tnmax	Maximum daily minimum temperature (tnmax)
cf.tnmean	Mean daily minimum temperature (tnmean)
cf.tnmin	Minimum daily minimum temperature (tnmin)
cf.tnn	Minimum daily minimum temperature (tnn)
cf.tnx	Maximum daily minimum temperature (tnx)
cf.tx	Mean of daily maximum temperature (tx)
cf.txm	Mean daily maximum temperature (txm)
cf.txmax	Maximum daily maximum temperature (txmax)
cf.txmean	Mean daily maximum temperature (txmean)
cf.txmin	Minimum daily maximum temperature (txmin)
cf.txn	Minimum daily maximum temperature (txn)
cf.txx	Maximum daily maximum temperature (txx)
cf.vdtr	Mean day-to-day variation in Diurnal Temperature Range (vdtr)
cold_and_dry_days	Cold and dry days
cold_and_wet_days	cold and wet days
cold_spell_days	Number of days part of a cold spell
cold_spell_duration_index	Number of days part of a percentile-defined cold spell (csdi_{window})
cold_spell_frequency	Number of cold spell events
consecutive_frost_days	Maximum number of consecutive days with Tmin < {thresh}
consecutive_frost_free_days	Maximum number of consecutive days with Tmin >= {thresh}
continuous_snow_cover_end	End date of continuous snow cover
continuous_snow_cover_start	Start date of continuous snow cover
cool_night_index	cool night index
cooling_degree_days	Cooling degree days (Tmean > {thresh})
corn_heat_units	Corn heat units (Tmin > {thresh_tasmin} and

(continues on next page)

(continued from previous page)

	Tmax > {thresh_tasmax}). (chu)
cwd	Maximum consecutive wet days (Precip >= {thresh})
days_over_precip_doy_thresh	Count of days with daily precipitation above the given percentile [days].
days_over_precip_thresh	Count of days with daily precipitation above the given percentile [days].
days_with_snow	Number of days with solid precipitation flux between low and high thresholds.
dc	Drought Code
degree_days_exceedance_date	Day of year when cumulative degree days exceed {sum_thresh}.
dlyfrzthw	daily freezethaw cycles
doy_qmax	Day of the year of the maximum over {indexer} (q{indexer}_doy_qmax)
doy_qmin	Day of the year of the minimum over {indexer} (q{indexer}_doy_qmin)
dry_days	Number of dry days (precip < {thresh})
dry_spell_frequency	The {freq} number of dry periods of minimum {window} days.
dry_spell_total_length	The {freq} total number of days in dry periods of minimum {window} days.
dtr	Mean Diurnal Temperature Range
dtrmax	Maximum Diurnal Temperature Range
dtrvar	Mean Diurnal Temperature Range Variability
e_sat	Saturation vapor pressure
effective_growing_degree_days	Effective growing degree days computed with {method} formula (Summation of max((Tmin + Tmax)/2 - {thresh}, 0), for days between between dynamically-determined start and end dates). (egdd)
etr	Intra-period Extreme Temperature Range
fire_season	Fire season mask
first_day_above	First day of year with temperature above {thresh}
first_day_below	First day of year with temperature below {thresh}
first_snowfall	Date of first snowfall
fit	{dist} distribution parameters (params)
fraction_over_precip_doy_thresh	Fraction of precipitation over threshold during wet days.
fraction_over_precip_thresh	Fraction of precipitation over threshold during wet days.
freezethaw_spell_frequency	{freq} number of freeze-thaw spells.
freezethaw_spell_max_length	{freq} maximal length of freeze-thaw spells.
freezethaw_spell_mean_length	{freq} average length of freeze-thaw spells.
freezing_degree_days	Freezing degree days (Tmean < {thresh})
freq_analysis	N-year return period {mode} {indexer} {window}-day flow (q{window}{mode:r}{indexer})
freshet_start	Day of year of spring freshet start

(continues on next page)

(continued from previous page)

frost_days	Number of frost days (Tmin < {thresh})
frost_free_season_end	Day of year of frost free season end
frost_free_season_length	Length of the frost free season
frost_free_season_start	Day of year of frost free season start
frost_season_length	Length of the frost season
fwi	Drought Code, Duff Moisture Code, Fine Fuel Moisture Code, Initial Spread Index, Buildup Index, Fire Weather Index (dc, dmc, ffmc, isi, bui, fwi)
growing_degree_days	Growing degree days above {thresh}
growing_season_end	Day of year of growing season end
growing_season_length	ETCCDI Growing Season Length (Tmean > {thresh})
growing_season_start	Day of year of growing season start
heat_index	heat index
heat_wave_frequency	Number of heat wave events (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax} for >= {window} days)
heat_wave_index	Number of days that are part of a heatwave
heat_wave_max_length	Maximum length of heat wave events (Tmin > {thresh_tasmin}and Tmax > {thresh_tasmax} for >= {window} days)
heat_wave_total_length	Total length of heat wave events (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax} for >= {window} days)
heating_degree_days	Heating degree days (Tmean < {thresh})
high_precip_low_temp	Count of days with high precipitation and low temperatures.
hot_spell_frequency	Number of hot spell events (Tmax > {thresh_tasmax} for >= {window} days)
hot_spell_max_length	Maximum length of hot spell events (Tmax > {thresh_tasmax} for >= {window} days)
huglin_index	Huglin heliothermal index (Summation of ((Tmin + Tmax)/2 - {thresh}) * Latitude-based day-lengthcoefficient (`k`), for days between {start_date} and {end_date}). (hi)
humidex	humidex index
hurs	Relative Humidity
hurs_fromdewpoint	Relative Humidity (hurs)
huss	Specific Humidity
huss_fromdewpoint	Specific Humidity
icclim.bedd	Biologically effective growing degree days (Summation of min((max((Tmin + Tmax)/2 - {thresh_tasmin}, 0)), 9°C), for days between 1 April and 30 September) (BEDD)
icclim.cd	Cold and dry days (CD)
icclim.cdd	Maximum number of consecutive dry days (RR<1 mm) (CDD)
icclim.cfd	Maximum number of consecutive frost days (TN<0°C) (CFD)
icclim.csdi	Cold-spell duration index (CSDI)
icclim.csu	Maximum number of consecutive summer day

(continues on next page)

(continued from previous page)

	(CSU)
icclim.cw	cold and wet days (CW)
icclim.cwd	Maximum number of consecutive wet days (RR1 mm) (CWD)
icclim.dtr	Mean of diurnal temperature range (DTR)
icclim.etr	Intra-period extreme temperature range (ETR)
icclim.fd	Frost days (TN<0°C) (FD)
icclim.gd4	Growing degree days (sum of TG>4°C) (GD4)
icclim.gsl	Growing season length (GSL)
icclim.hd17	Heating degree days (sum of17°C - TG) (HD17)
icclim.hi	Huglin heliothermal index (Summation of ((Tmean + Tmax)/2 - {thresh}) * Latitude-based day-length coefficient (`k`), for days between 1 April and 31 October) (HI)
icclim.id	Ice days (TX<0°C) (ID)
icclim.prcptot	Precipitation sum over wet days (PRCPTOT)
icclim.r10mm	Heavy precipitation days (precipitation10 mm) (R10mm)
icclim.r20mm	Very heavy precipitation days (precipitation20 mm) (R20mm)
icclim.r75p	Count of days with daily precipitation above the given percentile [days]. (days_over_precip_thresh)
icclim.r75ptot	Precipitation fraction due to moderate wet days (>75th percentile) (R75pTOT)
icclim.r95p	Count of days with daily precipitation above the given percentile [days]. (days_over_precip_thresh)
icclim.r95ptot	Precipitation fraction due to very wet days (>95th percentile) (R95pTOT)
icclim.r99p	Count of days with daily precipitation above the given percentile [days]. (days_over_precip_thresh)
icclim.r99ptot	Precipitation fraction due to extremely wet days (>99th percentile) (R99pTOT)
icclim.rr	Precipitation sum (RR)
icclim.rr1	Wet days (RR1 mm) (RR1)
icclim.rx1day	Highest 1-day precipitation amount (RX1day)
icclim.rx5day	Highest 5-day precipitation amount (RX5day)
icclim.sd	Mean of daily snow depth (SD)
icclim.sd1	Snow days (SD1 cm) (SD1)
icclim.sd50cm	Snow days (SD50 cm) (SD50cm)
icclim.sd5cm	Snow days (SD5 cm) (SD5cm)
icclim.sdii	Average precipitation during wet days (SDII) (sdii)
icclim.su	Summer days (TX>25°C) (SU)
icclim.tg	Mean daily mean temperature (tg_mean)
icclim.tg10p	Days with TG<10th percentile of daily mean temperature (cold days) (TG10p)
icclim.tg90p	Days with TG>90th percentile of daily mean temperature (warm days) (TG90p)
icclim.tgn	Minimum daily mean temperature (tg_min)

(continues on next page)

(continued from previous page)

icclim.tgx	Maximum daily mean temperature (tg_max)
icclim.tn	Mean daily minimum temperature (tn_mean)
icclim.tn10p	Days with TN<10th percentile of daily minimum temperature (cold nights) (TN10p)
icclim.tn90p	Days with TN>90th percentile of daily minimum temperature (warm nights) (TN90p)
icclim.tnn	Minimum daily minimum temperature (tn_min)
icclim.tnx	Maximum daily minimum temperature (tn_max)
icclim.tr	Tropical nights (TN>20°C) (TR)
icclim.tx	Mean daily maximum temperature (tx_mean)
icclim.tx10p	Days with TX<10th percentile of daily maximum temperature (cold day-times) (TX10p)
icclim.tx90p	Days with TX>90th percentile of daily maximum temperature (warm day-times) (TX90p)
icclim.txn	Minimum daily maximum temperature (tx_min)
icclim.txx	Maximum daily maximum temperature (tx_max)
icclim.vdtr	Mean absolute day-to-day difference in DTR (vDTR)
icclim.wd	Warm and dry days (WD)
icclim.wsdi	Warm-spell duration index (WSDI)
icclim.ww	Warm and wet days (WW)
ice_days	Number of ice days (Tmax < {thresh})
jetstream_metric_woollings	Latitude of maximum smoothed zonal wind speed, Maximum strength of smoothed zonal wind speed (jetlat, jetstr)
last_snowfall	Date of last snowfall
last_spring_frost	Day of year of last spring frost
latitude_temperature_index	Latitude-temperature index (lti)
liquid_precip_ratio	Ratio of rainfall to total precipitation.
liquidprcptot	Total liquid precipitation
max_n_day_precipitation_amount	maximum {window}-day total precipitation (rx{window}day)
max_pr_intensity	Maximum precipitation intensity over {window}h duration
maximum_consecutive_warm_days	The maximum number of days with tasmax > thresh per periods (summer days).
mean_radiant_temperature	Mean radiant temperature (mrt)
melt_and_precip_max	The maximum snow melt plus precipitation over a given number of days for each period. [mass/area]. ({freq}_melt_and_precip_max)
potential_evapotranspiration	Potential evapotranspiration (evspsblpot)
prcptot	Total precipitation
prlp	Liquid precipitation
prsn	Solid precipitation
rain_frzgr	Number of rain on frozen ground days
rb_flashiness_index	Richards-Baker flashiness index (rbi)
rprctot	The proportion of the total precipitation accounted for by convective precipitation for each period.
rx1day	maximum 1-day total precipitation
sdii	Average precipitation during wet days (SDII)
sea_ice_area	Sea ice area

(continues on next page)

(continued from previous page)

sea_ice_extent	Sea ice extent
snd_max_doy	Date when snow depth reaches its maximum value. ({freq}_snd_max_doy)
snow_cover_duration	Number of days with snow depth above threshold
snow_depth	Mean of daily snow depth
snow_melt_we_max	The maximum snow melt over a given number of days for each period. [mass/area]. ({freq}_snow_melt_we_max)
snw_max	Maximum daily snow amount ({freq}_snw_max)
snw_max_doy	Day of year of maximum daily snow amount ({freq}_snw_max_doy)
solidprcptot	Total solid precipitation
stats	{freq} {op} of {indexer} daily flow (q{indexer}{op:r})
tg	Daily mean temperature
tg10p	Number of days when Tmean < {tas_per_thresh}th percentile
tg90p	Number of days when Tmean > {tas_per_thresh}th percentile
tg_days_above	Number of days with Tavg > {thresh}
tg_days_below	Number of days with Tavg < {thresh}
tg_max	Maximum daily mean temperature
tg_mean	Mean daily mean temperature
tg_min	Minimum daily mean temperature
thawing_degree_days	Thawing degree days (degree days above 0°C)
tn10p	Number of days when Tmin < {tasmin_per_thresh}th percentile
tn90p	Number of days when Tmin > {tasmin_per_thresh}th percentile
tn_days_above	Number of days with Tmin > {thresh}
tn_days_below	Number of days with Tmin < {thresh}
tn_max	Maximum daily minimum temperature
tn_mean	Mean daily minimum temperature
tn_min	Minimum daily minimum temperature
tropical_nights	Number of Tropical Nights (Tmin > {thresh})
tx10p	Number of days when Tmax < {tasmax_per_thresh}th percentile
tx90p	Number of days when Tmax > {tasmax_per_thresh}th percentile
tx_days_above	Number of days with Tmax > {thresh}
tx_days_below	Number of days with Tmax < {thresh}
tx_max	Maximum daily maximum temperature
tx_mean	Mean daily maximum temperature
tx_min	Minimum daily maximum temperature
tx_tn_days_above	Number of days with Tmax > {thresh_tasmax} and Tmin > {thresh_tasmin}
utci	Universal Thermal Climate Index
warm_and_dry_days	warm and dry days
warm_and_wet_days	warm and wet days
warm_spell_duration_index	Number of days part of a percentile-defined warm spell

(continues on next page)

(continued from previous page)

```
water_budget                    Water budget
water_budget_from_tas           Water budget
wet_prcptot                     Total precipitation
wetdays                         Number of wet days (precip >= {thresh})
wetdays_prop                    Proportion of wet days (precip >= {thresh})
wind_chill                      Wind chill index
wind_speed_from_vector          Near-Surface Wind Speed, Near-Surface Wind
                                from Direction (sfcWind, sfcWindfromdir)
wind_vector_from_speed          Near-Surface Eastward Wind, Near-Surface
                                Northward Wind (uas, vas)
windy_days                      Number of days with surface wind speed above
                                threshold
winter_storm                    Number of days per period identified as
                                winter storms. ({freq}_winter_storm)
```

For more information about a specific indicator, you can either use the `info` subcommand or directly access the `--help` message of the indicator. The former gives more information about the metadata while the latter only prints the usage. Note that the module name (`atmos`, `land` or `seaIce`) is mandatory.

[3]:
```
!xclim info liquidprcptot
```

```
Indicator liquidprcptot:
    identifier : liquidprcptot
    title : Accumulated liquid precipitation.
    abstract : Resample the original daily mean precipitation flux
  and accumulate over each period. If a daily temperature is provided, the
  `phase` keyword can be used to sum precipitation of a given phase only. When
  the temperature is under the provided threshold, precipitation is assumed to
  be snow, and liquid rain otherwise. This indice is agnostic to the type of
  daily temperature (tas, tasmax or tasmin) given.
    keywords :
    outputs (#1)
      standard_name : lwe_thickness_of_liquid_precipitation_amount
      long_name : Total liquid precipitation
      units : mm
      cell_methods : time: sum over days
      description : Annual total liquid precipitation, estimated as
  precipitation when temperature >= 0 degc
      var_name : liquidprcptot
    notes : Let :math:`PR_i` be the mean daily precipitation of day
  :math:`i`, then for a period :math:`j` starting at day :math:`a` and
  finishing on day :math:`b`:

  .. math::

     PR_{ij} = \sum_{i=a}^{b} PR_i

  If tas and phase are given, the corresponding phase precipitation is
  estimated before computing the accumulation, using one of
  `snowfall_approximation` or `rain_approximation` with the `binary` method.
```

(continues on next page)

(continued from previous page)

```
Options:
  --pr VAR_NAME   Mean daily precipitation flux.  [default: pr]
  --tas VAR_NAME  Mean, maximum or minimum daily temperature.  [default: tas]
  --thresh TEXT   Threshold of `tas` over which the precipication is assumed
                  to be liquid rain.  [default: 0 degC]
  --freq TEXT     Resampling frequency.  [default: YS]
  --help          Show this message and exit.
```

In the usage message, VAR_NAME indicates that the passed argument must match a variable in the input dataset.

[4]:
```python
from __future__ import annotations

import warnings

import numpy as np
import pandas as pd
import xarray as xr
from pandas.plotting import register_matplotlib_converters

register_matplotlib_converters()
warnings.filterwarnings("ignore", "implicitly registered datetime converter")
%matplotlib inline
xr.set_options(display_style="html")

time = pd.date_range("2000-01-01", periods=366)
tasmin = xr.DataArray(
    -5 * np.cos(2 * np.pi * time.dayofyear / 365) + 273.15,
    dims=("time"),
    coords={"time": time},
    attrs={"units": "K"},
)
tasmax = xr.DataArray(
    -5 * np.cos(2 * np.pi * time.dayofyear / 365) + 283.15,
    dims=("time"),
    coords={"time": time},
    attrs={"units": "K"},
)
pr = xr.DataArray(
    np.clip(10 * np.sin(18 * np.pi * time.dayofyear / 365), 0, None),
    dims=("time"),
    coords={"time": time},
    attrs={"units": "mm/d"},
)
ds = xr.Dataset({"tasmin": tasmin, "tasmax": tasmax, "pr": pr})
ds.to_netcdf("example_data.nc")
```

8.1 Computing indicators

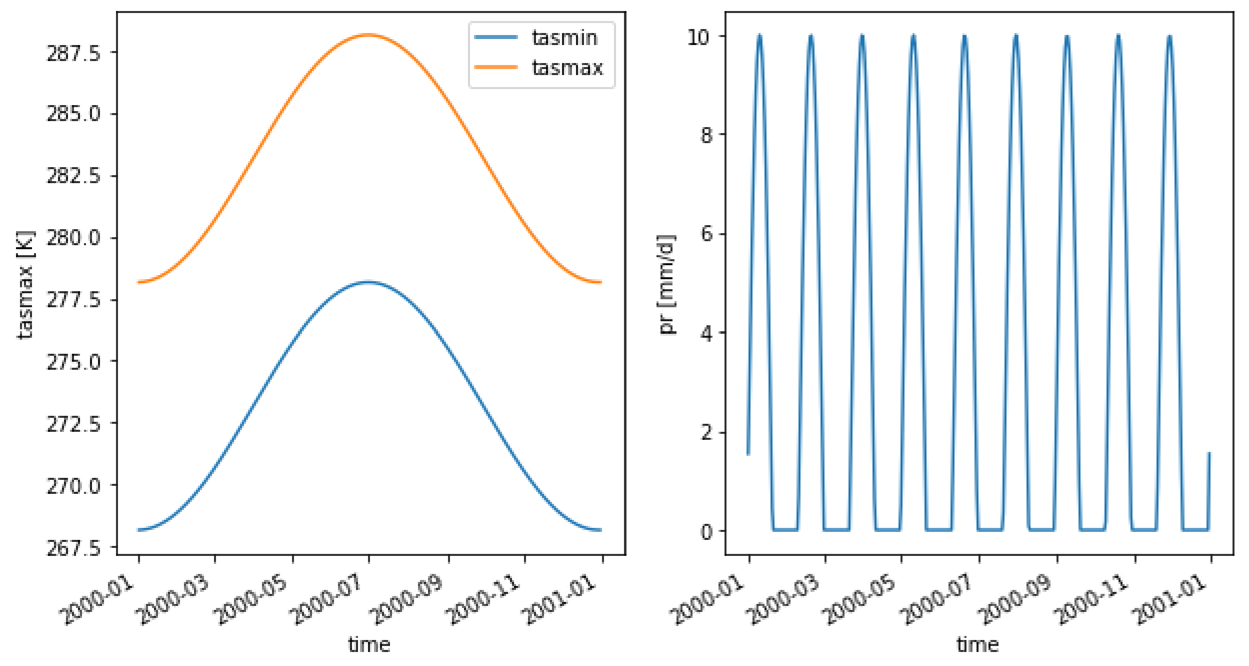
So let's say we have the following toy dataset:

[5]:
```
import xarray as xr

ds = xr.open_dataset("example_data.nc")
display(ds)
```

```
<xarray.Dataset>
Dimensions:  (time: 366)
Coordinates:
  * time     (time) datetime64[ns] 2000-01-01 2000-01-02 ... 2000-12-31
Data variables:
    tasmin   (time) float64 ...
    tasmax   (time) float64 ...
    pr       (time) float64 ...
```

[6]:
```
import matplotlib.pyplot as plt

fig, (axT, axpr) = plt.subplots(1, 2, figsize=(10, 5))
ds.tasmin.plot(label="tasmin", ax=axT)
ds.tasmax.plot(label="tasmax", ax=axT)
ds.pr.plot(ax=axpr)
axT.legend()
```

[6]:
```
<matplotlib.legend.Legend at 0x7f7e78600820>
```



To compute an indicator, say the monthly solid precipitation accumulation, we simply call:

[7]:
```
!xclim -i example_data.nc -o out1.nc solidprcptot --pr pr --tas tasmin --freq MS
```

```
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/core/cfchecks.py:41: UserWarning: Variable does not have a `cell_
↪methods` attribute.
  _check_cell_methods(
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/core/cfchecks.py:45: UserWarning: Variable does not have a `standard_
↪name` attribute.
  check_valid(vardata, "standard_name", data["standard_name"])
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/indicators/atmos/_precip.py:80: UserWarning: Variable does not have a␣
↪`standard_name` attribute.
  cfchecks.check_valid(tas, "standard_name", "air_temperature")
[########################################] | 100% Completed |  0.1s
```

In this example, we decided to use `tasmin` for the `tas` variable. We didn't need to provide the `--pr` parameter as our data has the same name.

Finally, more than one indicators can be computed to the output dataset by simply chaining the calls:

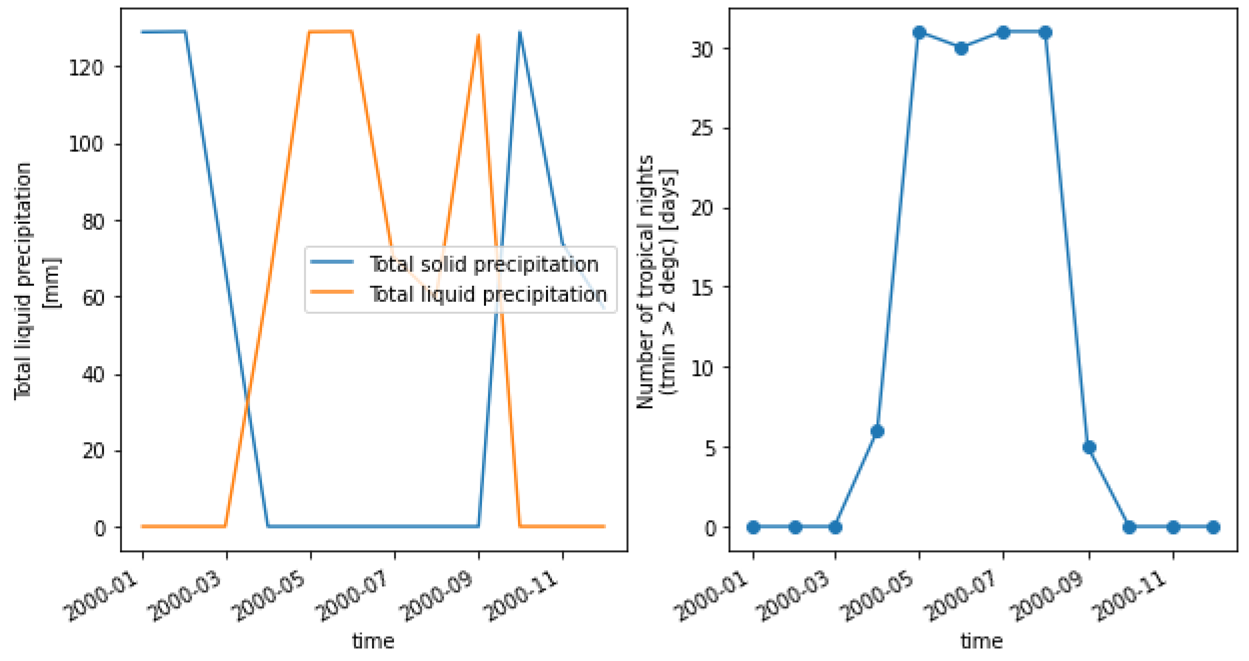
[8]:
```
!xclim -i example_data.nc -o out2.nc liquidprcptot --tas tasmin --freq MS tropical_
↪nights --thresh "2 degC" --freq MS
```

```
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/core/cfchecks.py:41: UserWarning: Variable does not have a `cell_
↪methods` attribute.
  _check_cell_methods(
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/core/cfchecks.py:45: UserWarning: Variable does not have a `standard_
↪name` attribute.
  check_valid(vardata, "standard_name", data["standard_name"])
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/indicators/atmos/_precip.py:80: UserWarning: Variable does not have a␣
↪`standard_name` attribute.
  cfchecks.check_valid(tas, "standard_name", "air_temperature")
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/core/cfchecks.py:41: UserWarning: Variable does not have a `cell_
↪methods` attribute.
  _check_cell_methods(
/home/docs/checkouts/readthedocs.org/user_builds/xclim/envs/v0.37.0/lib/python3.8/site-
↪packages/xclim/core/cfchecks.py:45: UserWarning: Variable does not have a `standard_
↪name` attribute.
  check_valid(vardata, "standard_name", data["standard_name"])
[########################################] | 100% Completed |  0.1s
```

Let's see the outputs:

[9]:
```
ds1 = xr.open_dataset("out1.nc")
ds2 = xr.open_dataset("out2.nc", decode_timedelta=False)

fig, (axPr, axTn) = plt.subplots(1, 2, figsize=(10, 5))
ds1.solidprcptot.plot(ax=axPr, label=ds1.solidprcptot.long_name)
ds2.liquidprcptot.plot(ax=axPr, label=ds2.liquidprcptot.long_name)
ds2.tropical_nights.plot(ax=axTn, marker="o")
axPr.legend()
```

[9]:
```
<matplotlib.legend.Legend at 0x7f7e43d32220>
```



[10]:
```
ds1.close()
```

[11]:
```
ds2.close()
```

8.2 Data Quality Checks

As of version 0.30.0, `xclim` now also provides a command-line utility for performing data quality control checks on existing NetCDF files.

These checks examine the values of data_variables for suspicious value patterns (e.g. values that repeat for many days) or erroneous values (e.g. humidity percentages outside of 0-100, minimum temperatures exceeding maximum temperatures, etc.). The checks (called "data flags") are based on the ECAD ICCLIM quality control checks (https://www.ecad.eu/documents/atbd.pdf).

The full list of checks performed for each variable are listed in `xclim/core/data/variables.yml`.

[12]:
```
!xclim dataflags --help
```

```
Usage: xclim dataflags [OPTIONS] [VARIABLES]...

  Run quality control checks on input data variables and flag for quality
  control issues or suspicious values.

Options:
  -r, --raise-flags  Print an exception in the event that a variable is found
                     to have quality control issues.
  -a, --append       Return the netCDF dataset with the `ecad_qc_flag` array
                     appended as a data_var.
```

(continues on next page)

(continued from previous page)

```
  -d, --dims TEXT    Dimensions upon which aggregation should be performed.
                     Default: "all". Ignored if no variable provided.
  -f, --freq TEXT    Resampling periods frequency used for aggregation.
                     Default: None. Ignored if no variable provided.
  --help             Show this message and exit.
```

When running the dataflags CLI checks, you must either set an output file (`-o filename.nc`) or set the checks to raise if there are any failed checks (`-r`).

By default, when setting an output file, the returned file will only contain the flag value (`True` if no flags were raised, `False` otherwise). To append the flag to a copy of the dataset, we use the `-a` option.

The default behaviour is to raise a flag if any element of the array resolves to `True` (ie: aggregated across all dimensions), but we can specify the level of aggregation by dimension with the `-d` or `--dims` option.

[13]:
```
# Create an output file with just the flag value and no aggregation (dims=None)

!xclim -i example_data.nc -o flag_output.nc dataflags -d none

# Need to wait until the file is written

!sleep 2s
```

```
[########################################] | 100% Completed |  2.4s
```

[14]:
```
import xarray as xr

ds1 = xr.open_dataset("flag_output.nc")
display(ds1.data_vars, ds1.ecad_qc_flag)
ds1.close()
```

```
Data variables:
    ecad_qc_flag  (time) bool ...
```

```
<xarray.DataArray 'ecad_qc_flag' (time: 366)>
array([ True,  True,  True, ...,  True,  True,  True])
Coordinates:
  * time     (time) datetime64[ns] 2000-01-01 2000-01-02 ... 2000-12-31
Attributes:
    comment:  Adheres to ECAD quality control checks.
    history:  [2022-06-18 02:28:17] - xclim version: 0.37.0 - Performed the f...
```

[15]:
```
# Create an output file with values appended to the original dataset.

!xclim -i example_data.nc -o flag_output_appended.nc dataflags -a

# Need to wait until the file is written
!sleep 2s
```

```
[########################################] | 100% Completed |  1.6s
```

[16]:
```
import xarray as xr

ds2 = xr.open_dataset("flag_output_appended.nc")
```

(continues on next page)

(continued from previous page)

```
display(ds2.data_vars, ds2.ecad_qc_flag)
ds2.close()
```

```
Data variables:
    tasmin        (time) float64 ...
    tasmax        (time) float64 ...
    pr            (time) float64 ...
    ecad_qc_flag  bool ...
```

```
<xarray.DataArray 'ecad_qc_flag' ()>
array(True)
Attributes:
    comment:  Adheres to ECAD quality control checks.
    history:  [2022-06-18 02:28:39] - xclim version: 0.37.0 - Performed the f...
```

[17]:
```
# Raise an error if any quality control checks fail. Passing example:

!xclim -i example_data.nc dataflags -r
```

```
Dataset passes quality control checks!
```

[18]:
```
import xarray as xr

# Create some bad data with minimum temperatures exceeding max temperatures
bad_ds = xr.open_dataset("example_data.nc")

# Swap entire variable arrays
bad_ds["tasmin"].values, bad_ds["tasmax"].values = (
    bad_ds.tasmax.values,
    bad_ds.tasmin.values,
)
bad_ds.to_netcdf("suspicious_data.nc")
bad_ds.close()
```

[19]:
```
# Raise an error if any quality control checks fail. Failing example:

!xclim -i suspicious_data.nc dataflags -r
```

```
Data quality flags indicate suspicious values. Flags raised are:
  - Maximum temperature values found below minimum temperatures.
  - Maximum temperature values found below minimum temperatures.
```

These checks can also be set to examine a specifc variable within a netcdf file, with more descriptive information for each check performed.

[20]:
```
!xclim -i example_data.nc -o flag_output_pr.nc dataflags pr
```

```
[########################################] | 100% Completed |  2.3s
```

[21]:
```
import xarray as xr

ds3 = xr.open_dataset("flag_output_pr.nc")
display(ds3.data_vars)
```

(continues on next page)

(continued from previous page)

```
for dv in ds3.data_vars:
    display(ds3[dv])
```

```
Data variables:
    negative_accumulation_values               bool ...
    very_large_precipitation_events            bool ...
    values_eq_5_repeating_for_5_or_more_days   bool ...
    values_eq_1_repeating_for_10_or_more_days  bool ...
```

```
<xarray.DataArray 'negative_accumulation_values' ()>
array(False)
Attributes:
    description:  Negative values found for pr.
    units:
    history:      [2022-06-18 02:29:39] pr: negative_accumulation_values(da=p...
```

```
<xarray.DataArray 'very_large_precipitation_events' ()>
array(False)
Attributes:
    description:  Precipitation events in excess of 300 mm d-1 for pr.
    units:
    history:      [2022-06-18 02:29:39] pr: very_large_precipitation_events(d...
```

```
<xarray.DataArray 'values_eq_5_repeating_for_5_or_more_days' ()>
array(False)
Attributes:
    description:  Repetitive values at 5.0 for at least 5 days found for pr.
    units:
    history:      [2022-06-18 02:29:39] pr: values_op_thresh_repeating_for_n_...
```

```
<xarray.DataArray 'values_eq_1_repeating_for_10_or_more_days' ()>
array(False)
Attributes:
    description:  Repetitive values at 1.0 for at least 10 days found for pr.
    units:
    history:      [2022-06-18 02:29:39] pr: values_op_thresh_repeating_for_n_...
```

CHAPTER

NINE

BIAS ADJUSTMENT AND DOWNSCALING ALGORITHMS

xarray data structures allow for relatively straightforward implementations of simple bias-adjustment and downscaling algorithms documented in *Adjustment Methods*. Each algorithm is split into *train* and *adjust* components. The *train* function will compare two DataArrays x and y, and create a dataset storing the *transfer* information allowing to go from x to y. This dataset, stored in the adjustment object, can then be used by the *adjust* method to apply this information to x. x could be the same *DataArray* used for training, or another *DataArray* with similar characteristics.

For example, given a daily time series of observations *ref*, a model simulation over the observational period *hist* and a model simulation over a future period *sim*, we would apply a bias-adjustment method such as *detrended quantile mapping* (DQM) as:

```
from xclim import sdba
dqm = sdba.adjustment.DetrendedQuantileMapping.train(ref, hist)
scen = dqm.adjust(sim)
```

Most method can either be applied additively or multiplicatively. Also, most methods can be applied independently on different time groupings (monthly, seasonally) or according to the day of the year and a rolling window width.

When transfer factors are applied in adjustment, they can be interpolated according to the time grouping. This helps avoid discontinuities in adjustment factors at the beginning of each season or month and is computationally cheaper than computing adjustment factors for each day of the year. (Currently only implemented for monthly grouping)

9.1 Application in multivariate settings

When applying univariate adjustment methods to multiple variables, some strategies are recommended to avoid introducing unrealistic artifacts in adjusted outputs.

9.1.1 Minimum and maximum temperature

When adjusting both minimum and maximum temperature, adjustment factors sometimes yield minimum temperatures larger than the maximum temperature on the same day, which of course, is nonsensical. One way to avoid this is to first adjust maximum temperature using an additive adjustment, then adjust the diurnal temperature range (DTR) using a multiplicative adjustment, and then determine minimum temperature by subtracting DTR from the maximum temperature ([Thrasher2012], [AgbazoGrenier2019])

9.1.2 Relative and specific humidity

When adjusting both relative and specific humidity, we want to preserve the relationship between both. To do this, [Grenier2018] suggests to first adjust the relative humidity using a multiplicative factor, ensure values are within 0-100%, then apply an additive adjustment factor to the surface pressure before estimating the specific humidity from thermodynamic relationships.

9.1.3 Radiation and precipitation

In theory, short wave radiation should be capped when precipitation is not zero, but there is as of yet no mechanism proposed to do that, see [HoffmanRath2012].

9.2 References

9.3 SDBA User API

9.3.1 Adjustment Methods

`class xclim.sdba.adjustment.DetrendedQuantileMapping(`**args*, *_trained=False*, ***kwargs*`)`

Bases: `TrainAdjust`

Detrended Quantile Mapping bias-adjustment.

The algorithm follows these steps, 1-3 being the ‘train’ and 4-6, the ‘adjust’ steps.

1. A scaling factor that would make the mean of *hist* match the mean of *ref* is computed.
2. *ref* and *hist* are normalized by removing the “dayofyear” mean.
3. Adjustment factors are computed between the quantiles of the normalized *ref* and *hist*.
4. *sim* is corrected by the scaling factor, and either normalized by “dayofyear” and detrended group-wise or directly detrended per “dayofyear”, using a linear fit (modifiable).
5. Values of detrended *sim* are matched to the corresponding quantiles of normalized *hist* and corrected accordingly.
6. The trend is put back on the result.

$$F^{-1}_{ref}\left\{F_{hist}\left[\frac{\overline{hist}\cdot sim}{\overline{sim}}\right]\right\}\frac{\overline{sim}}{\overline{hist}}$$

where F is the cumulative distribution function (CDF) and $\overline{xyz}$ is the linear trend of the data. This equation is valid for multiplicative adjustment. Based on the DQM method of [Cannon2015].

Parameters

- **Train step**
- **nquantiles** (*int or 1d array of floats*) – The number of quantiles to use. See *equally_spaced_nodes()*. An array of quantiles [0, 1] can also be passed. Defaults to 20 quantiles.
- **kind** ({‘+’, ‘’}*) – The adjustment kind, either additive or multiplicative. Defaults to “+”.

- **group** (*Union[str, Grouper]*) – The grouping information. See `xclim.sdba.base.Grouper` for details. Default is "time", meaning an single adjustment group along dimension "time".
- **Adjust step**
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use when interpolating the adjustment factors. Defaults to "nearest".
- **detrend** (*int or BaseDetrend instance*) – The method to use when detrending. If an int is passed, it is understood as a PolyDetrend (polynomial detrending) degree. Defaults to 1 (linear detrending)
- **extrapolation** (*{'constant', 'nan'}*) – The type of extrapolation to use. See `xclim.sdba.utils.extrapolate_qm()` for details. Defaults to "constant".

References

`class xclim.sdba.adjustment.EmpiricalQuantileMapping(`**args*, *_trained=False*, ***kwargs*`)`

Bases: `TrainAdjust`

Empirical Quantile Mapping bias-adjustment.

Adjustment factors are computed between the quantiles of *ref* and *sim*. Values of *sim* are matched to the corresponding quantiles of *hist* and corrected accordingly.

$$F^{-1}_{ref}(F_{hist}(sim))$$

where F is the cumulative distribution function (CDF) and *mod* stands for model data.

Parameters

- **Train step**
- **nquantiles** (*int or 1d array of floats*) – The number of quantiles to use. Two endpoints at 1e-6 and 1 - 1e-6 will be added. An array of quantiles [0, 1] can also be passed. Defaults to 20 quantiles.
- **kind** (*{'+', ''}**) – The adjustment kind, either additive or multiplicative. Defaults to "+".
- **group** (*Union[str, Grouper]*) – The grouping information. See `xclim.sdba.base.Grouper` for details. Default is "time", meaning an single adjustment group along dimension "time".
- **Adjust step**
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use when interpolating the adjustment factors. Defaults to "nearset".
- **extrapolation** (*{'constant', 'nan'}*) – The type of extrapolation to use. See `xclim.sdba.utils.extrapolate_qm()` for details. Defaults to "constant".

References

Dequé, M. (2007). Frequency of precipitation and temperature extremes over France in an anthropogenic scenario: Model results and statistical correction according to observed values. Global and Planetary Change, 57(1–2), 16–26. https://doi.org/10.1016/j.gloplacha.2006.11.030

_allow_diff_calendars = False

class xclim.sdba.adjustment.ExtremeValues(*args*, *_trained=False*, ***kwargs*)

Bases: TrainAdjust

Adjustment correction for extreme values.

The tail of the distribution of adjusted data is corrected according to the bias between the parametric Generalized Pareto distributions of the simulatated and reference data, [RRJF2021]. The distributions are composed of the maximal values of clusters of "large" values. With "large" values being those above *cluster_thresh*. Only extreme values, whose quantile within the pool of large values are above *q_thresh*, are re-adjusted. See Notes.

This adjustment method should be considered experimental and used with care.

Parameters

- **Train step**
- **cluster_thresh** (*Quantity (str with units)*) – The threshold value for defining clusters.
- **q_thresh** (*float*) – The quantile of "extreme" values, [0, 1[. Defaults to 0.95.
- **ref_params** (*xr.DataArray, optional*) – Distribution parameters to use instead of fitting a GenPareto distribution on *ref*.
- **Adjust step**
- **scen** (*DataArray*) – This is a second-order adjustment, so the adjust method needs the first-order adjusted timeseries in addition to the raw "sim".
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use when interpolating the adjustment factors. Defaults to "linear".
- **extrapolation** (*{'constant', 'nan'}*) – The type of extrapolation to use. See extrapolate_qm() for details. Defaults to "constant".
- **frac** (*float*) – Fraction where the cutoff happens between the original scen and the corrected one. See Notes,]0, 1]. Defaults to 0.25.
- **power** (*float*) – Shape of the correction strength, see Notes. Defaults to 1.0.

Notes

Extreme values are extracted from *ref*, *hist* and *sim* by finding all "clusters", i.e. runs of consecutive values above *cluster_thresh*. The *q_thresh`th percentile of these values is taken on `ref* and *hist* and becomes *thresh*, the extreme value threshold. The maximal value of each cluster, if it exceeds that new threshold, is taken and Generalized Pareto distributions are fitted to them, for both *ref* and *hist*. The probabilities associated with each of these extremes in *hist* is used to find the corresponding value according to *ref*'s distribution. Adjustment factors are computed as the bias between those new extremes and the original ones.

In the adjust step, a Generalized Pareto distributions is fitted on the cluster-maximums of *sim* and it is used to associate a probability to each extreme, values over the *thresh* compute in the training, without the clustering. The adjustment factors are computed by interpolating the trained ones using these probabilities and the probabilities computed from *hist*.

Finally, the adjusted values (C_i) are mixed with the pre-adjusted ones (*scen*, D_i) using the following transition function:

$$V_i = C_i * \tau + D_i * (1 - \tau)$$

Where τ is a function of sim's extreme values (unadjusted, S_i) and of arguments `frac` (f) and `power` (p):

$$\tau = \left(\frac{1}{f}\frac{S - min(S)}{max(S) - min(S)}\right)^p$$

Code based on an internal Matlab source and partly ib the *biascorrect_ extremes* function of the julia package [ClimateTools].

Because of limitations imposed by the lazy computing nature of the dask backend, it is not possible to know the number of cluster extremes in *ref* and *hist* at the moment the output data structure is created. This is why the code tries to estimate that number and usually overestimates it. In the training dataset, this translated into a *quantile* dimension that is too large and variables *af* and *px_ hist* are assigned NaNs on extra elements. This has no incidence on the calculations themselves but requires more memory than is useful.

References

class xclim.sdba.adjustment.LOCI(**args*, *_trained=False*, ***kwargs*)

Bases: `TrainAdjust`

Local Intensity Scaling (LOCI) bias-adjustment.

This bias adjustment method is designed to correct daily precipitation time series by considering wet and dry days separately ([Schmidli2006]).

Multiplicative adjustment factors are computed such that the mean of *hist* matches the mean of *ref* for values above a threshold.

The threshold on the training target *ref* is first mapped to *hist* by finding the quantile in *hist* having the same exceedance probability as thresh in *ref*. The adjustment factor is then given by

$$s = \frac{\langle ref : ref \geq t_{ref} \rangle - t_{ref}}{\langle hist : hist \geq t_{hist} \rangle - t_{hist}}$$

In the case of precipitations, the adjustment factor is the ratio of wet-days intensity.

For an adjustment factor s, the bias-adjustment of *sim* is:

$$sim(t) = \max\left(t_{ref} + s \cdot (hist(t) - t_{hist}), 0\right)$$

Parameters

- **Train step**
- **group** (*Union[str, Grouper]*) – The grouping information. See `xclim.sdba.base.Grouper` for details. Default is "time", meaning an single adjustment group along dimension "time".
- **thresh** (*str*) – The threshold in *ref* above which the values are scaled.
- **Adjust step**
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use then interpolating the adjustment factors. Defaults to "linear".

References

`class xclim.sdba.adjustment.NpdfTransform(*args, _trained=False, **kwargs)`

Bases: `Adjust`

N-dimensional probability density function transform.

This adjustment object combines both training and adjust steps in the *adjust* class method.

A multivariate bias-adjustment algorithm described by [Cannon2018], as part of the MBCn algorithm, based on a color-correction algorithm described by [Pitie2005].

This algorithm in itself, when used with QuantileDeltaMapping, is NOT trend-preserving. The full MBCn algorithm includes a reordering step provided here by `xclim.sdba.processing.reordering()`.

See notes for an explanation of the algorithm.

Parameters

- **base** (*BaseAdjustment*) – An univariate bias-adjustment class. This is untested for anything else than QuantileDeltaMapping.
- **base_kws** (*dict, optional*) – Arguments passed to the training of the univariate adjustment.
- **n_escore** (*int*) – The number of elements to send to the escore function. The default, 0, means all elements are included. Pass -1 to skip computing the escore completely. Small numbers result in less significative scores, but the execution time goes up quickly with large values.
- **n_iter** (*int*) – The number of iterations to perform. Defaults to 20.
- **pts_dim** (*str*) – The name of the "multivariate" dimension. Defaults to "multivar", which is the normal case when using `xclim.sdba.base.stack_variables()`.
- **adj_kws** (*dict, optional*) – Dictionary of arguments to pass to the adjust method of the univariate adjustment.
- **rot_matrices** (*xr.DataArray, optional*) – The rotation matrices as a 3D array ('iterations', <pts_dim>, <anything>), with shape (n_iter, <N>, <N>). If left empty, random rotation matrices will be automatically generated.

Notes

The historical reference (T, for "target"), simulated historical (H) and simulated projected (S) datasets are constructed by stacking the timeseries of N variables together. The algorithm is broken into the following steps:

1. Rotate the datasets in the N-dimensional variable space with **R**, a random rotation NxN matrix.

..math

```
\tilde{\mathbf{T}} = \mathbf{T}\mathbf{R} \
\tilde{\mathbf{H}} = \mathbf{H}\mathbf{R} \
\tilde{\mathbf{S}} = \mathbf{S}\mathbf{R}
```

2. An univariate bias-adjustment $\mathcal{F}$ is used on the rotated datasets. The adjustments are made in additive mode, for each variable i.

$$\hat{\mathbf{H}}_i, \hat{\mathbf{S}}_i = \mathcal{F}\left(\tilde{\mathbf{T}}_i, \tilde{\mathbf{H}}_i, \tilde{\mathbf{S}}_i\right)$$

3. The bias-adjusted datasets are rotated back.

$$\mathbf{H}' = \hat{\mathbf{H}}\mathbf{R}$$

$$\mathbf{S}' = \hat{\mathbf{S}}\mathbf{R}$$

These three steps are repeated a certain number of times, prescribed by argument `n_iter`. At each iteration, a new random rotation matrix is generated.

The original algorithm ([Pitie2005]), stops the iteration when some distance score converges. Following [Cannon2018] and the MBCn implementation in [CannonR], we instead fix the number of iterations.

As done by [Cannon2018], the distance score chosen is the "Energy distance" from [SkezelyRizzo2004] (see `xclim.sdba.processing.escore()`).

The random matrices are generated following a method laid out by [Mezzadri2006].

This is only part of the full MBCn algorithm, see *Statistical Downscaling and Bias-Adjustment* for an example on how to replicate the full method with xclim. This includes a standardization of the simulated data beforehand, an initial univariate adjustment and the reordering of those adjusted series according to the rank structure of the output of this algorithm.

References

`class xclim.sdba.adjustment.PrincipalComponents(*args, _trained=False, **kwargs)`

Bases: `TrainAdjust`

Principal component adjustment.

This bias-correction method maps model simulation values to the observation space through principal components ([Hnilica2017]). Values in the simulation space (multiple variables, or multiple sites) can be thought of as coordinates along axes, such as variable, temperature, etc. Principal components (PC) are a linear combinations of the original variables where the coefficients are the eigenvectors of the covariance matrix. Values can then be expressed as coordinates along the PC axes. The method makes the assumption that bias-corrected values have the same coordinates along the PC axes of the observations. By converting from the observation PC space to the original space, we get bias corrected values. See notes for a mathematical explanation.

Note that *principal components* is meant here as the algebraic operation defining a coordinate system based on the eigenvectors, not statistical principal component analysis.

Parameters

- **group** (*Union[str, Grouper]*) – The main dimension and grouping information. See Notes. See `xclim.sdba.base.Grouper` for details. The adjustment will be performed on each group independently. Default is "time", meaning an single adjustment group along dimension "time".
- **best_orientation** (*{'simple', 'full'}*) – Which method to use when searching for the best principal component orientation. See `best_pc_orientation_simple()` and `best_pc_orientation_full()`. "full" is more precise, but it is much slower.
- **crd_dim** (*str*) – The data dimension along which the multiple simulation space dimensions are taken. For a multivariate adjustment, this usually is "multivar", as returned by *sdba.stack_variables*. For a multisite adjustment, this should be the spatial dimension. The training algorithm currently doesn't support any chunking along either *crd_dim*. *group.dim* and *group.add_dims*.

Notes

The input data is understood as a set of N points in a M-dimensional space.

- M is taken along *crd_dim*.
- N is taken along the dimensions given through *group* : (the main *dim* but also, if requested, the *add_dims* and *window*).

The principal components (PC) of *hist* and *ref* are used to defined new coordinate systems, centered on their respective means. The training step creates a matrix defining the transformation from *hist* to *ref*:

$$scen = e_R + \mathbf{T}(sim - e_H)$$

Where:

$$\mathbf{T} = \mathbf{R}\mathbf{H}^{-1}$$

R is the matrix transforming from the PC coordinates computed on *ref* to the data coordinates. Similarly, **H** is transform from the *hist* PC to the data coordinates (**H** is the inverse transformation). e_R and e_H are the centroids of the *ref* and *hist* distributions respectively. Upon running the *adjust* step, one may decide to use e_S, the centroid of the *sim* distribution, instead of e_H.

References

`class xclim.sdba.adjustment.QuantileDeltaMapping(*args, _trained=False, **kwargs)`

Bases: `EmpiricalQuantileMapping`

Quantile Delta Mapping bias-adjustment.

Adjustment factors are computed between the quantiles of *ref* and *hist*. Quantiles of *sim* are matched to the corresponding quantiles of *hist* and corrected accordingly.

$$sim\frac{F^{-1}_{ref}\left[F_{sim}(sim)\right]}{F^{-1}_{hist}\left[F_{sim}(sim)\right]}$$

where F is the cumulative distribution function (CDF). This equation is valid for multiplicative adjustment. The algorithm is based on the “QDM” method of [Cannon2015].

Parameters

- **Train step**
- **nquantiles** (*int or 1d array of floats*) – The number of quantiles to use. See `equally_spaced_nodes()`. An array of quantiles [0, 1] can also be passed. Defaults to 20 quantiles.
- **kind** (*{'+', ''}**) – The adjustment kind, either additive or multiplicative. Defaults to “+”.
- **group** (*Union[str, Grouper]*) – The grouping information. See `xclim.sdba.base.Grouper` for details. Default is “time”, meaning an single adjustment group along dimension “time”.
- **Adjust step**
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use when interpolating the adjustment factors. Defaults to “nearest”.

- **extrapolation** (*{'constant', 'nan'}*) – The type of extrapolation to use. See `xclim.sdba.utils.extrapolate_qm()` for details. Defaults to "constant".
- **Extra diagnostics**
- ——————
- **In adjustment**
- **quantiles** (The quantile of each value of *sim*. The adjustment factor is interpolated using this as the "quantile" axis on *ds.af*.)

References

`class xclim.sdba.adjustment.Scaling(`**args*, *_trained=False*, ***kwargs*`)`

Bases: `TrainAdjust`

Scaling bias-adjustment.

Simple bias-adjustment method scaling variables by an additive or multiplicative factor so that the mean of *hist* matches the mean of *ref*.

Parameters

- **Train step**
- **group** (*Union[str, Grouper]*) – The grouping information. See *`xclim.sdba.base.Grouper`* for details. Default is "time", meaning an single adjustment group along dimension "time".
- **kind** (*{'+', ''}**) – The adjustment kind, either additive or multiplicative. Defaults to "+".
- **Adjust step**
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use then interpolating the adjustment factors. Defaults to "nearest".

9.3.2 Pre and post processing

`xclim.sdba.processing.adapt_freq(`*ref: xr.DataArray*, *sim: xr.DataArray*, ***, *group:* Grouper | *str*, *thresh: str = '0 mm d-1'*`)` → xr.Dataset

Adapt frequency of values under thresh of *sim*, in order to match ref.

This is useful when the dry-day frequency in the simulations is higher than in the references. This function will create new non-null values for *sim/hist*, so that adjustment factors are less wet-biased. Based on [Themessl2012].

Parameters

- **ds** (*xr.Dataset*) – With variables : "ref", Target/reference data, usually observed data. and "sim", Simulated data.
- **dim** (*str*) – Dimension name.
- **group** (*Union[str, Grouper]*) – Grouping information, see base.Grouper
- **thresh** (*str*) – Threshold below which values are considered zero, a quantity with units.

Returns

- **sim_adj** (*xr.DataArray*) – Simulated data with the same frequency of values under threshold than ref. Adjustment is made group-wise.
- **pth** (*xr.DataArray*) – For each group, the smallest value of sim that was not frequency-adjusted. All values smaller were either left as zero values or given a random value between thresh and pth. NaN where frequency adaptation wasn't needed.
- **dP0** (*xr.DataArray*) – For each group, the percentage of values that were corrected in sim.

Notes

With P_0^r the frequency of values under threshold T_0 in the reference (ref) and P_0^s the same for the simulated values,
$DeltaP_0 =$
$fracP_0^s - P_0^r P_0^s$, when positive, represents the proportion of values under T_0 that need to be corrected.

The correction replaces a proportion
$DeltaP_0$ of the values under T_0 in sim by a uniform random number between T_0 and P_{th}, where $P_{th} = F_{ref}^{-1}(F_{sim}(T_0))$ and *F(x)* is the empirical cumulative distribution function (CDF).

References

`xclim.sdba.processing.construct_moving_yearly_window(`*da: Dataset*, *window: int = 21*, *step: int = 1*, *dim: str = 'movingwin'*`)`

Construct a moving window DataArray.

Stacks windows of *da* in a new 'movingwin' dimension. Windows are always made of full years, so calendar with non-uniform year lengths are not supported.

Windows are constructed starting at the beginning of *da*, if number of given years is not a multiple of *step*, then the last year(s) will be missing as a supplementary window would be incomplete.

Parameters

- **da** (*xr.Dataset*) – A DataArray with a *time* dimension.
- **window** (*int*) – The length of the moving window as a number of years.
- **step** (*int*) – The step between each window as a number of years.
- **dim** (*str*) – The new dimension name. If given, must also be given to *unpack_moving_yearly_window*.

Returns

xr.DataArray – A DataArray with a new *movingwin* dimension and a *time* dimension with a length of 1 window. This assumes downstream algorithms do not make use of the _absolute_ year of the data. The correct timeseries can be reconstructed with `unpack_moving_yearly_window()`. The coordinates of *movingwin* are the first date of the windows.

`xclim.sdba.processing.escore(`*tgt: DataArray*, *sim: DataArray*, *dims: Sequence[str] = ('variables', 'time')*, *N: int = 0*, *scale: bool = False*`)` → DataArray

Energy score, or energy dissimilarity metric, based on [SzekelyRizzo] and [Cannon18].

Parameters

- **tgt** (*xr.DataArray*) – Target observations.

- **sim** (*xr.DataArray*) – Candidate observations. Must have the same dimensions as *tgt*.
- **dims** (*sequence of 2 strings*) – The name of the dimensions along which the variables and observation points are listed. *tgt* and *sim* can have different length along the second one, but must be equal along the first one. The result will keep all other dimensions.
- **N** (*int*) – If larger than 0, the number of observations to use in the score computation. The points are taken evenly distributed along *obs_dim*.
- **scale** (*bool*) – Whether to scale the data before computing the score. If True, both arrays as scaled according to the mean and standard deviation of *tgt* along *obs_dim*. (std computed with *ddof=1* and both statistics excluding NaN values).

Returns

xr.DataArray – e-score with dimensions not in *dims*.

Notes

Explanation adapted from the "energy" R package documentation. The e-distance between two clusters C_i, C_j (tgt and sim) of size n_i, n_j proposed by Székely and Rizzo (2004) is defined by:

$$e(C_i, C_j) = \frac{1}{2}\frac{n_i n_j}{n_i + n_j}\left[2M_{ij} M_{ii} M_{jj}\right]$$

where

$$M_{ij} = \frac{1}{n_i n_j}\sum_{p=1}^{n_i}\sum_{q=1}^{n_j}\left\|X_{ip} Xjq\right\|.$$

$\|\cdot\|$ denotes Euclidean norm, X_{ip} denotes the p-th observation in the i-th cluster.

The input scaling and the factor $\frac{1}{2}$ in the first equation are additions of [Cannon18] to the metric. With that factor, the test becomes identical to the one defined by [BaringhausFranz]. This version is tested against values taken from Alex Cannon's MBC R package.

References

`xclim.sdba.processing.from_additive_space(`*data: DataArray, lower_bound: Optional[str] = None, upper_bound: Optional[str] = None, trans: Optional[str] = None, units: Optional[str] = None*`)`

Transform back to the physical space a variable that was transformed with *to_addtitive_space*.

Based on [AlavoineGrenier]. If parameters are not present on the attributes of the data, they must be all given are arguments.

Parameters

- **data** (*xr.DataArray*) – A variable that was transform by `to_additive_space()`.
- **lower_bound** (*str, optional*) – The smallest physical value of the variable, as a Quantity string. The final data will have no value smaller or equal to this bound. If None (default), the *sdba_transform_lower* attribute is looked up on *data*.
- **upper_bound** (*str, optional*) – The largest physical value of the variable, as a Quantity string. Only relevant for the logit transformation. The final data will have no value larger or equal to this bound. If None (default), the *sdba_transform_upper* attribute is looked up on *data*.

- **trans** (*{'log', 'logit'}, optional*) – The transformation to use. See notes. If None (the default), the *sdba_transform* attribute is looked up on *data*.
- **units** (*str, optional*) – The units of the data before transformation to the additive space. If None (the default), the *sdba_transform_units* attribute is looked up on *data*.

Returns

xr.DataArray – The physical variable. Attributes are conserved, even if some might be incorrect. Except units which are taken from *sdba_transform_units* if available. All *sdba_transform** attributes are deleted.

Notes

Given a variable that is not usable in an additive adjustment, `to_additive_space()` applied a transformation to a space where additive methods are sensible. Given Y the transformed variable, b_- the lower physical bound of that variable and b_+ the upper physical bound, two back-transformations are currently implemented to get X, the physical variable.

- *log*

$$X = e^Y + b_-$$

- *logit*

$$X' = \frac{1}{1+e^{-Y}} X = X * (b_+ - b_-) + b_-$$

See also:

`to_additive_space`
for the original transformation.

References

`xclim.sdba.processing.jitter(`*x: xr.DataArray, lower: str | None = None, upper: str | None = None, minimum: str | None = None, maximum: str | None = None*`)` → xr.DataArray

Replace values under a threshold and values above another by a uniform random noise.

Not to be confused with R's *jitter*, which adds uniform noise instead of replacing values.

Parameters

- **x** (*xr.DataArray*) – Values.
- **lower** (*str, optional*) – Threshold under which to add uniform random noise to values, a quantity with units. If None, no jittering is performed on the lower end.
- **upper** (*str, optional*) – Threshold over which to add uniform random noise to values, a quantity with units. If None, no jittering is performed on the upper end.
- **minimum** (*str, optional*) – Lower limit (excluded) for the lower end random noise, a quantity with units. If None but *lower* is not None, 0 is used.

- **maximum** (*str, optional*) – Upper limit (excluded) for the upper end random noise, a quantity with units. If *upper* is not None, it must be given.

Returns
xr.DataArray – Same as x but values < lower are replaced by a uniform noise in range (minimum, lower) and values >= upper are replaced by a uniform noise in range [upper, maximum). The two noise distributions are independent.

`xclim.sdba.processing.jitter_over_thresh`(*x: DataArray*, *thresh: str*, *upper_bnd: str*) → DataArray

Replace values greater than threshold by a uniform random noise.

Do not confuse with R's jitter, which adds uniform noise instead of replacing values.

Parameters

- **x** (*xr.DataArray*) – Values.
- **thresh** (*str*) – Threshold over which to add uniform random noise to values, a quantity with units.
- **upper_bnd** (*str*) – Maximum possible value for the random noise, a quantity with units.

Returns
xr.DataArray

Notes

If thresh is low, this will change the mean value of x.

`xclim.sdba.processing.jitter_under_thresh`(*x: DataArray*, *thresh: str*) → DataArray

Replace values smaller than threshold by a uniform random noise.

Do not confuse with R's jitter, which adds uniform noise instead of replacing values.

Parameters

- **x** (*xr.DataArray*) – Values.
- **thresh** (*str*) – Threshold under which to add uniform random noise to values, a quantity with units.

Returns
xr.DataArray

Notes

If thresh is high, this will change the mean value of x.

`xclim.sdba.processing.normalize`(*data: xr.DataArray*, *norm: xr.DataArray | None = None*, ***, *group: Grouper | str*, *kind: str = '+'*) → xr.Dataset

Normalize an array by removing its mean.

Normalization if performed group-wise and according to *kind*.

Parameters

- **data** (*xr.DataArray*) – The variable to normalize.
- **norm** (*xr.DataArray, optional*) – If present, it is used instead of computing the norm again.

- **group** (*Union[str, Grouper]*) – Grouping information. See `xclim.sdba.base.Grouper` for details..
- **kind** (*{'+', ''}**) – If *kind* is "+", the mean is subtracted from the mean and if it is '*', it is divided from the data.

Returns

- *xr.DataArray* – Groupwise anomaly.
- **norm** (*xr.DataArray*) – Mean over each group.

`xclim.sdba.processing.reordering(`*ref: DataArray*, *sim: DataArray*, *group: str = 'time'*`)` → Dataset

Reorders data in *sim* following the order of ref.

The rank structure of *ref* is used to reorder the elements of *sim* along dimension "time", optionally doing the operation group-wise.

Parameters

- **sim** (*xr.DataArray*) – Array to reorder.
- **ref** (*xr.DataArray*) – Array whose rank order sim should replicate.
- **group** (*str*) – Grouping information. See `xclim.sdba.base.Grouper` for details.

Returns

- *xr.Dataset* – sim reordered according to ref's rank order.
- *Reference*
- *———*
- **.. [Cannon18] Cannon, A. J. (2018). Multivariate quantile mapping bias correction** (*An N-dimensional probability density function transform for climate model simulations of multiple variables. Climate Dynamics, 50(1), 31–49. https://doi.org/10.1007/s00382-017-3580-6*)

`xclim.sdba.processing.stack_variables(`*ds: Dataset*, *rechunk: bool = True*, *dim: str = 'multivar'*`)`

Stack different variables of a dataset into a single DataArray with a new "variables" dimension.

Variable attributes are all added as lists of attributes to the new coordinate, prefixed with "_". Variables are concatenated in the new dimension in alphabetical order, to ensure coherent behaviour with different datasets.

Parameters

- **ds** (*xr.Dataset*) – Input dataset.
- **rechunk** (*bool*) – If True (default), dask arrays are rechunked with *variables : -1*.
- **dim** (*str*) – Name of dimension along which variables are indexed.

Returns

xr.DataArray – The transformed variable. Attributes are conserved, even if some might be incorrect. Except units, which are replaced with "". Old units are stored in *sdba_transformation_units*. A *sdba_transform* attribute is added, set to the transformation method. *sdba_transform_lower* and *sdba_transform_upper* are also set if the requested bounds are different from the defaults.

Array with variables stacked along *dim* dimension. Units are set to "".

`xclim.sdba.processing.standardize(`*da: xr.DataArray, mean: xr.DataArray | None = None, std: xr.DataArray | None = None, dim: str = 'time'*`)` → tuple[xr.DataArray | xr.Dataset, xr.DataArray, xr.DataArray]

Standardize a DataArray by centering its mean and scaling it by its standard deviation.

Either of both of mean and std can be provided if need be.

Returns the standardized data, the mean and the standard deviation.

`xclim.sdba.processing.to_additive_space(`*data: DataArray, lower_bound: str, upper_bound: Optional[str] = None, trans: str = 'log'*`)`

Transform a non-additive variable into an additive space by the means of a log or logit transformation.

Based on [AlavoineGrenier].

Parameters

- **data** (*xr.DataArray*) – A variable that can't usually be bias-adusted by additive methods.
- **lower_bound** (*str*) – The smallest physical value of the variable, excluded, as a Quantity string. The data should only have values strictly larger than this bound.
- **upper_bound** (*str, optional*) – The largest physical value of the variable, excluded, as a Quantity string. Only relevant for the logit transformation. The data should only have values strictly smaller than this bound.
- **trans** (*{'log', 'logit'}*) – The transformation to use. See notes.

Notes

Given a variable that is not usable in an additive adjustment, this apply a transformation to a space where additive methods are sensible. Given X the variable, b_- the lower physical bound of that variable and b_+ the upper physical bound, two transformations are currently implemented to get Y, the additive-ready variable. ln is the natural logarithm.

- *log*

$$Y = \ln\left(X - b_- \right)$$

  Usually used for variables with only a lower bound, like precipitation (*pr*, *prsn*, etc) and daily temperature range (*dtr*). Both have a lower bound of 0.

- *logit*

$$X' = (X - b_-) / (b_+ - b_-) Y = \ln\left(\frac{X'}{1 - X'}\right)$$

  Usually used for variables with both a lower and a upper bound, like relative and specific humidity, cloud cover fraction, etc.

This will thus produce *Infinity* and *NaN* values where $X == b_-$ or $X == b_+$. We recommend using *`jitter_under_thresh()`* and *`jitter_over_thresh()`* to remove those issues.

See also:

`from_additive_space`
  for the inverse transformation.

jitter_under_thresh
Remove values exactly equal to the lower bound.

jitter_over_thresh
Remove values exactly equal to the upper bound.

References

`xclim.sdba.processing.uniform_noise_like`(*da: DataArray*, *low: float = 1e-06*, *high: float = 0.001*) → DataArray

Return a uniform noise array of the same shape as da.

Noise is uniformly distributed between low and high. Alternative method to *jitter_under_thresh* for avoiding zeroes.

`xclim.sdba.processing.unpack_moving_yearly_window`(*da: DataArray*, *dim: str = 'movingwin'*, *append_ends: bool = True*)

Unpack a constructed moving window dataset to a normal timeseries, only keeping the central data.

Unpack DataArrays created with `construct_moving_yearly_window()` and recreate a timeseries data. If *append_ends* is False, only keeps the central non-overlapping years. The final timeseries will be (window - step) years shorter than the initial one. If *append_ends* is True, the time points from first and last windows will be included in the final timeseries.

The time points that are not in a window will never be included in the final timeseries. The window length and window step are inferred from the coordinates.

Parameters

- **da** (*xr.DataArray*) – As constructed by `construct_moving_yearly_window()`.
- **dim** (*str*) – The window dimension name as given to the construction function.
- **append_ends** (*bool*) – Whether to append the ends of the timeseries If False, the final timeseries will be (window - step) years shorter than the initial one, but all windows will contribute equally. If True, the year before the middle years of the first window and the years after the middle years of the last window are appended to the middle years. The final timeseries will be the same length as the initial timeseries if the windows span the whole timeseries. The time steps that are not in a window will be left out of the final timeseries.

`xclim.sdba.processing.unstack_variables`(*da: DataArray*, *dim: Optional[str] = None*)

Unstack a DataArray created by *stack_variables* to a dataset.

Parameters

- **da** (*xr.DataArray*) – Array holding different variables along *dim* dimension.
- **dim** (*str*) – Name of dimension along which the variables are stacked. If not specified (default), *dim* is inferred from attributes of the coordinate.

Returns

xr.Dataset – Dataset holding each variable in an individual DataArray.

`xclim.sdba.processing.unstandardize`(*da: DataArray*, *mean: DataArray*, *std: DataArray*)

Rescale a standardized array by performing the inverse operation of *standardize*.

9.3.3 Detrending Objects

class xclim.sdba.detrending.LoessDetrend(*group='time'*, *kind='+'*, *f=0.2*, *niter=1*, *d=0*, *weights='tricube'*, *equal_spacing=None*, *skipna=True*)

Bases: `BaseDetrend`

Detrend time series using a LOESS regression.

The fit is a piecewise linear regression. For each point, the contribution of all neighbors is weighted by a bell-shaped curve (gaussian) with parameters sigma (std). The x-coordinate of the DataArray is scaled to [0,1] before the regression is computed.

Parameters

- **group** (*Union[str, Grouper]*) – The grouping information. See `xclim.sdba.base.Grouper` for details. The fit is performed along the group's main dim.
- **kind** (*{'*', '+'}*) – The way the trend is removed or added, either additive or multiplicative.
- **d** (*[0, 1]*) – Order of the local regression. Only 0 and 1 currently implemented.
- **f** (*float*) – Parameter controlling the span of the weights, between 0 and 1.
- **niter** (*int*) – Number of robustness iterations to execute.
- **weights** (*["tricube", "gaussian"]*) – Shape of the weighting function: "tricube" : a smooth top-hat like curve, f gives the span of non-zero values. "gaussian" : a gaussian curve, f gives the span for 95% of the values.
- **skipna** (*bool*) – If True (default), missing values are not included in the loess trend computation and thus are not propagated. The output will have the same missing values as the input.

Notes

LOESS smoothing is computationally expensive. As it relies on a loop on gridpoints, it can be useful to use smaller than usual chunks. Moreover, it suffers from heavy boundary effects. As a rule of thumb, the outermost N * f/2 points should be considered dubious. (N is the number of points along each group)

class xclim.sdba.detrending.MeanDetrend(***, *group: Grouper | str = 'time'*, *kind: str = '+'*, ***kwargs*)

Bases: `BaseDetrend`

Simple detrending removing only the mean from the data, quite similar to normalizing.

class xclim.sdba.detrending.NoDetrend(***, *group: Grouper | str = 'time'*, *kind: str = '+'*, ***kwargs*)

Bases: `BaseDetrend`

Convenience class for polymorphism. Does nothing.

class xclim.sdba.detrending.PolyDetrend(*group='time'*, *kind='+'*, *degree=4*, *preserve_mean=False*)

Bases: `BaseDetrend`

Detrend time series using a polynomial regression.

Parameters

- **group** (*Union[str, Grouper]*) – The grouping information. See `xclim.sdba.base.Grouper` for details. The fit is performed along the group's main dim.

- **kind** (*{'', '+'}**) – The way the trend is removed or added, either additive or multiplicative.
- **degree** (*int*) – The order of the polynomial to fit.
- **preserve_mean** (*bool*) – Whether to preserve the mean when de/re-trending. If True, the trend has its mean removed before it is used.

`class xclim.sdba.detrending.RollingMeanDetrend`(*group='time'*, *kind='+'*, *win=30*, *weights=None*, *min_periods=None*)

Bases: *BaseDetrend*

Detrend time series using a rolling mean.

Parameters

- **group** (*Union[str, Grouper]*) – The grouping information. See *xclim.sdba.base.Grouper* for details. The fit is performed along the group's main dim.
- **kind** (*{'', '+'}**) – The way the trend is removed or added, either additive or multiplicative.
- **win** (*int*) – The size of the rolling window. Units are the steps of the grouped data, which means this detrending is best use with either *group='time'* or *group='time.dayofyear'*. Other grouping will have large jumps included within the windows and :py`:class:*LoessDetrend* might offer a better solution.
- **weights** (*sequence of floats, optional*) – Sequence of length *win*. Defaults to None, which means a flat window.
- **min_periods** (*int, optional*) – Minimum number of observations in window required to have a value, otherwise the result is NaN. See `xarray.DataArray.rolling()`. Defaults to None, which sets it equal to *win*. Setting both *weights* and this is not implemented yet.

Notes

As for the *LoessDetrend* detrending, important boundary effects are to be expected.

9.3.4 Statistical Downscaling and Bias Adjustment Utilities

`xclim.sdba.utils.add_cyclic_bounds`(*da: xr.DataArray*, *att: str*, *cyclic_coords: bool = True*) → xr.DataArray | xr.Dataset

Reindex an array to include the last slice at the beginning and the first at the end.

This is done to allow interpolation near the end-points.

Parameters

- **da** (*Union[xr.DataArray, xr.Dataset]*) – An array
- **att** (*str*) – The name of the coordinate to make cyclic
- **cyclic_coords** (*bool*) – If True, the coordinates are made cyclic as well, if False, the new values are guessed using the same step as their neighbour.

Returns

Union[xr.DataArray, xr.Dataset] – da but with the last element along att prepended and the last one appended.

`xclim.sdba.utils.apply_correction`(*x: xr.DataArray*, *factor: xr.DataArray*, *kind: str | None = None*) → xr.DataArray

Apply the additive or multiplicative correction/adjustment factors.

If kind is not given, default to the one stored in the "kind" attribute of factor.

`xclim.sdba.utils.best_pc_orientation_full`(*R: ndarray*, *Hinv: ndarray*, *Rmean: ndarray*, *Hmean: ndarray*, *hist: ndarray*) → ndarray

Return best orientation vector for A according to the method of Alavoine et al. (2021, preprint).

Eigenvectors returned by *pc_matrix* do not have a defined orientation. Given an inverse transform Hinv, a transform R, the actual and target origins Hmean and Rmean and the matrix of training observations hist, this computes a scenario for all possible orientations and return the orientation that maximizes the Spearman correlation coefficient of all variables. The correlation is computed for each variable individually, then averaged.

This trick is explained in [alavoine2021]. See documentation of `sdba.adjustment.PrincipalComponentAdjustment()`.

Parameters

- **R** (*np.ndarray*) – MxM Matrix defining the final transformation.
- **Hinv** (*np.ndarray*) – MxM Matrix defining the (inverse) first transformation.
- **Rmean** (*np.ndarray*) – M vector defining the target distribution center point.
- **Hmean** (*np.ndarray*) – M vector defining the original distribution center point.
- **hist** (*np.ndarray*) – MxN matrix of all training observations of the M variables/sites.

Returns

np.ndarray – M vector of orientation correction (1 or -1).

References

`xclim.sdba.utils.best_pc_orientation_simple`(*R: ndarray*, *Hinv: ndarray*, *val: float = 1000*) → ndarray

Return best orientation vector according to a simple test.

Eigenvectors returned by *pc_matrix* do not have a defined orientation. Given an inverse transform Hinv and a transform R, this returns the orientation minimizing the projected distance for a test point far from the origin.

This trick is inspired by the one exposed in [hnilica2017]. For each possible orientation vector, the test point is reprojected and the distance from the original point is computed. The orientation minimizing that distance is chosen. See documentation of *sdba.adjustment.PrincipalComponentAdjustment*.

Parameters

- **R** (*np.ndarray*) – MxM Matrix defining the final transformation.
- **Hinv** (*np.ndarray*) – MxM Matrix defining the (inverse) first transformation.
- **val** (*float*) – The coordinate of the test point (same for all axes). It should be much greater than the largest furthest point in the array used to define B.

Returns

np.ndarray – Mx1 vector of orientation correction (1 or -1).

References

`xclim.sdba.utils.broadcast(`*grouped: xr.DataArray, x: xr.DataArray, *, group: str | Grouper = 'time', interp: str = 'nearest', sel: Mapping[str, xr.DataArray] | None = None*`)` → xr.DataArray

Broadcast a grouped array back to the same shape as a given array.

Parameters

- **grouped** (*xr.DataArray*) – The grouped array to broadcast like *x*.
- **x** (*xr.DataArray*) – The array to broadcast grouped to.
- **group** (*Union[str, Grouper]*) – Grouping information. See `xclim.sdba.base.Grouper` for details.
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use,
- **sel** (*Mapping[str, xr.DataArray]*) – Mapping of grouped coordinates to x coordinates (other than the grouping one).

Returns

xr.DataArray

`xclim.sdba.utils.copy_all_attrs(`*ds: xr.Dataset | xr.DataArray, ref: xr.Dataset | xr.DataArray*`)`

Copy all attributes of ds to ref, including attributes of shared coordinates, and variables in the case of Datasets.

`xclim.sdba.utils.ecdf(`*x: DataArray, value: float, dim: str = 'time'*`)` → DataArray

Return the empirical CDF of a sample at a given value.

Parameters

- **x** (*array*) – Sample.
- **value** (*float*) – The value within the support of *x* for which to compute the CDF value.
- **dim** (*str*) – Dimension name.

Returns

xr.DataArray – Empirical CDF.

`xclim.sdba.utils.ensure_longest_doy(`*func: Callable*`)` → Callable

Ensure that selected day is the longest day of year for x and y dims.

`xclim.sdba.utils.equally_spaced_nodes(`*n: int, eps: float | None = None*`)` → np.array

Return nodes with *n* equally spaced points within [0, 1], optionally adding two end-points.

Parameters

- **n** (*int*) – Number of equally spaced nodes.
- **eps** (*float, optional*) – Distance from 0 and 1 of added end nodes. If None (default), do not add endpoints.

Returns

np.array – Nodes between 0 and 1. Nodes can be seen as the middle points of *n* equal bins.

Warning: Passing a small *eps* will effectively clip the scenario to the bounds of the reference on the historical period in most cases. With normal quantile mapping algorithms, this can give strange result when the reference does not show as many extremes as the simulation does.

Notes

For n=4, eps=0 : 0—x——x——x——x—1

`xclim.sdba.utils.get_clusters(`*data: DataArray*, *u1*, *u2*, *dim: str = 'time'*`)` → Dataset

Get cluster count, maximum and position along a given dim.

See *get_clusters_1d*. Used by *adjustment.ExtremeValues*.

Parameters

- **data** (*1D ndarray*) – Values to get clusters from.
- **u1** (*float*) – Extreme value threshold, at least one value in the cluster must exceed this.
- **u2** (*float*) – Cluster threshold, values above this can be part of a cluster.
- **dim** (*str*) – Dimension name.

Returns

xr.Dataset –

With variables,

- *nclusters* : Number of clusters for each point (with *dim* reduced), int
- *start* : First index in the cluster (*dim* reduced, new *cluster*), int
- *end* : Last index in the cluster, inclusive (*dim* reduced, new *cluster*), int
- *maxpos* : Index of the maximal value within the cluster (*dim* reduced, new *cluster*), int
- *maximum* : Maximal value within the cluster (*dim* reduced, new *cluster*), same dtype as data.

For *start*, *end* and *maxpos*, -1 means NaN and should always correspond to a *NaN* in *maximum*. The length along *cluster* is half the size of "dim", the maximal theoretical number of clusters.

`xclim.sdba.utils.get_clusters_1d(`*data: np.ndarray*, *u1: float*, *u2: float*`)` → tuple[np.array, np.array, np.array, np.array]

Get clusters of a 1D array.

A cluster is defined as a sequence of values larger than u2 with at least one value larger than u1.

Parameters

- **data** (*1D ndarray*) – Values to get clusters from.
- **u1** (*float*) – Extreme value threshold, at least one value in the cluster must exceed this.
- **u2** (*float*) – Cluster threshold, values above this can be part of a cluster.

Returns

(np.array, np.array, np.array, np.array)

References

getcluster of Extremes.jl (read on 2021-04-20) https://github.com/jojal5/Extremes.jl

`xclim.sdba.utils.get_correction`(*x: DataArray*, *y: DataArray*, *kind: str*) → DataArray

Return the additive or multiplicative correction/adjustment factors.

`xclim.sdba.utils.interp_on_quantiles`(*newx: xr.DataArray*, *xq: xr.DataArray*, *yq: xr.DataArray*, ***, *group: str | Grouper = 'time'*, *method: str = 'linear'*, *extrapolation: str = 'constant'*)

Interpolate values of yq on new values of x.

Interpolate in 2D with `griddata()` if grouping is used, in 1D otherwise, with `interp1d`. Any NaNs in xq or yq are removed from the input map. Similarly, NaNs in newx are left NaNs.

Parameters

- **newx** (*xr.DataArray*) – The values at which to evaluate *yq*. If *group* has group information, *new* should have a coordinate with the same name as the group name In that case, 2D interpolation is used.
- **xq, yq** (*xr.DataArray*) – Coordinates and values on which to interpolate. The interpolation is done along the "quantiles" dimension if *group* has no group information. If it does, interpolation is done in 2D on "quantiles" and on the group dimension.
- **group** (*Union[str, Grouper]*) – The dimension and grouping information. (ex: "time" or "time.month"). Defaults to "time".
- **method** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method.
- **extrapolation** (*{'constant', 'nan'}*) – The extrapolation method used for values of *newx* outside the range of *xq*. See notes.

Notes

Extrapolation methods:

- 'nan' : Any value of *newx* outside the range of *xq* is set to NaN.
- 'constant' : Values of *newx* smaller than the minimum of *xq* are set to the first value of *yq* and those larger than the maximum, set to the last one (first and last non-nan values along the "quantiles" dimension). When the grouping is "time.month", these limits are linearly interpolated along the month dimension.

`xclim.sdba.utils.invert`(*x: xr.DataArray*, *kind: str | None = None*) → xr.DataArray

Invert a DataArray either additively (-x) or multiplicatively (1/x).

If kind is not given, default to the one stored in the "kind" attribute of x.

`xclim.sdba.utils.map_cdf`(*ds: Dataset*, ***, *y_value: DataArray*, *dim*)

Return the value in *x* with the same CDF as *y_value* in *y*.

This function is meant to be wrapped in a *Grouper.apply*.

Parameters

- **ds** (*xr.Dataset*) – Variables: x, Values from which to pick, y, Reference values giving the ranking
- **y_value** (*float, array*) – Value within the support of *y*.
- **dim** (*str*) – Dimension along which to compute quantile.

Returns
array – Quantile of x with the same CDF as y_value in y.

`xclim.sdba.utils.map_cdf_1d(`*x*, *y*, *y_value*`)`

Return the value in x with the same CDF as y_value in y.

`xclim.sdba.utils.pc_matrix(`*arr: np.ndarray | dsk.Array*`)` → np.ndarray | dsk.Array

Construct a Principal Component matrix.

This matrix can be used to transform points in arr to principal components coordinates. Note that this function does not manage NaNs; if a single observation is null, all elements of the transformation matrix involving that variable will be NaN.

Parameters
arr (*numpy.ndarray or dask.array.Array*) – 2D array (M, N) of the M coordinates of N points.

Returns
numpy.ndarray or dask.array.Array – MxM Array of the same type as arr.

`xclim.sdba.utils.rand_rot_matrix(`*crd: xr.DataArray*, *num: int = 1*, *new_dim: str | None = None*`)` → xr.DataArray

Generate random rotation matrices.

Rotation matrices are members of the SO(n) group, where n is the matrix size (*crd.size*). They can be characterized as orthogonal matrices with determinant 1. A square matrix R is a rotation matrix if and only if $R^t = R^1$ and $\det R = 1$.

Parameters

- **crd** (*xr.DataArray*) – 1D coordinate DataArray along which the rotation occurs. The output will be square with the same coordinate replicated, the second renamed to *new_dim*.
- **num** (*int*) – If larger than 1 (default), the number of matrices to generate, stacked along a "matrices" dimension.
- **new_dim** (*str*) – Name of the new "prime" dimension, defaults to the same name as *crd* + "_prime".

Returns
xr.DataArray – float, NxN if num = 1, numxNxN otherwise, where N is the length of crd.

References

Mezzadri, F. (2006). How to generate random matrices from the classical compact groups. arXiv preprint math-ph/0609050.

`xclim.sdba.utils.rank(`*da: DataArray*, *dim: str = 'time'*, *pct: bool = False*`)` → DataArray

Ranks data along a dimension.

Replicates *xr.DataArray.rank* but as a function usable in a Grouper.apply(). Xarray's docstring is below:

Equal values are assigned a rank that is the average of the ranks that would have been otherwise assigned to all the values within that set. Ranks begin at 1, not 0. If pct, computes percentage ranks.

Parameters

- **da** (*xr.DataArray*) – Source array.

- **dim** (*str, hashable*) – Dimension over which to compute rank.
- **pct** (*bool, optional*) – If True, compute percentage ranks, otherwise compute integer ranks.

Returns

DataArray – DataArray with the same coordinates and dtype 'float64'.

Notes

The *bottleneck* library is required. NaNs in the input array are returned as NaNs.

class `xclim.sdba.base.Grouper`(*group: str*, *window: int = 1*, *add_dims: Sequence[str] | set[str] | None = None*)

Create the Grouper object.

Parameters

- **group** (*str*) – The usual grouping name as xarray understands it. Ex: "time.month" or "time". The dimension name before the dot is the "main dimension" stored in *Grouper.dim* and the property name after is stored in *Grouper.prop*.
- **window** (*int*) – If larger than 1, a centered rolling window along the main dimension is created when grouping data. Units are the sampling frequency of the data along the main dimension.
- **add_dims** (*Optional[Union[Sequence[str], str]]*) – Additional dimensions that should be reduced in grouping operations. This behaviour is also controlled by the *main_only* parameter of the *apply* method. If any of these dimensions are absent from the dataarrays, they will be omitted.

`apply`(*func: FunctionType | str*, *da: xr.DataArray | Mapping[str, xr.DataArray] | xr.Dataset*, *main_only: bool = False*, ***kwargs*)

Apply a function group-wise on DataArrays.

Parameters

- **func** (*Union[FunctionType, str]*) – The function to apply to the groups, either a callable or a *xr.core.groupby.GroupBy* method name as a string. The function will be called as *func(group, dim=dims, **kwargs)*. See *main_only* for the behaviour of *dims*.
- **da** (*Union[xr.DataArray, Mapping[str, xr.DataArray], xr.Dataset]*) – The DataArray on which to apply the function. Multiple arrays can be passed through a dictionary. A dataset will be created before grouping.
- **main_only** (*bool*) – Whether to call the function with the main dimension only (if True) or with all grouping dims (if False, default) (including the window and dimensions given through *add_dims*). The dimensions used are also written in the "group_compute_dims" attribute. If all the input arrays are missing one of the 'add_dims', it is silently omitted.
- **kwargs** – Other keyword arguments to pass to the function.

Returns

DataArray or Dataset – Attributes "group", "group_window" and "group_compute_dims" are added.

If the function did not reduce the array:

- The output is sorted along the main dimension.
- The output is rechunked to match the chunks on the input If multiple inputs with differing chunking were given as inputs, the chunking with the smallest number of chunks is used.

If the function reduces the array:

- If there is only one group, the singleton dimension is squeezed out of the output
- The output is rechunked as to have only 1 chunk along the new dimension.

Notes

For the special case where a Dataset is returned, but only some of its variable where reduced by the grouping, xarray's *GroupBy.map* will broadcast everything back to the ungrouped dimensions. To overcome this issue, function may add a "_group_apply_reshape" attribute set to True on the variables that should be reduced and these will be re-grouped by calling *da.groupby(self.name).first()*.

`property freq`

Format a frequency string corresponding to the group.

For use with xarray's resampling functions.

`classmethod from_kwargs(`***kwargs*`)`

Parameterize groups using kwargs.

`get_coordinate(`*ds=None*`)`

Return the coordinate as in the output of group.apply.

Currently, only implemented for groupings with prop == *month* or *dayofyear*. For prop == *dayfofyear*, a ds (Dataset or DataArray) can be passed to infer the max day of year from the available years and calendar.

`get_index(`*da: xr.DataArray | xr.Dataset*, *interp: bool | None = None*`)`

Return the group index of each element along the main dimension.

Parameters

- **da** (*Union[xr.DataArray, xr.Dataset]*) – The input array/dataset for which the group index is returned. It must have Grouper.dim as a coordinate.
- **interp** (*bool, optional*) – If True, the returned index can be used for interpolation. Only value for month grouping, where integer values represent the middle of the month, all other days are linearly interpolated in between.

Returns

xr.DataArray – The index of each element along *Grouper.dim*. If *Grouper.dim* is *time* and *Grouper.prop* is None, an uniform array of True is returned. If *Grouper.prop* is a time accessor (month, dayofyear, etc), an numerical array is returned, with a special case of *month* and *interp=True*. If *Grouper.dim* is not *time*, the dim is simply returned.

`group(`*da: xr.DataArray | xr.Dataset = None*, *main_only=False*, ***das: xr.DataArray*`)`

Return a xr.core.groupby.GroupBy object.

More than one array can be combined to a dataset before grouping using the *das* kwargs. A new *window* dimension is added if *self.window* is larger than 1. If *Grouper.dim* is 'time', but 'prop' is None, the whole array is grouped together.

When multiple arrays are passed, some of them can be grouped along the same group as self. They are boadcasted, merged to the grouping dataset and regrouped in the output.

`property prop_name`

Create a significant name for the grouping.

9.3.5 Numba-accelerated utilities

`xclim.sdba.nbutils.quantile(`*da*, *q*, *dim*`)`

Compute the quantiles from a fixed list *q*.

`xclim.sdba.nbutils.remove_NaNs(`*x*`)`

Remove NaN values from series.

`xclim.sdba.nbutils.vecquantiles(`*da*, *rnk*, *dim*`)`

For when the quantile (rnk) is different for each point.

da and rnk must share all dimensions but dim.

9.3.6 LOESS Smoothing Module

`xclim.sdba.loess.loess_smoothing(`*da: xr.DataArray*, *dim: str = 'time'*, *d: int = 1*, *f: float = 0.5*, *niter: int = 2*, *weights: str | Callable = 'tricube'*, *equal_spacing: bool | None = None*, *skipna: bool = True*`)`

Locally weighted regression in 1D: fits a nonparametric regression curve to a scatter plot.

Returns a smoothed curve along given dimension. The regression is computed for each point using a subset of neighbouring points as given from evaluating the weighting function locally. Follows the procedure of [Cleveland1979].

Parameters

- **da** (*xr.DataArray*) – The data to smooth using the loess approach.
- **dim** (*str*) – Name of the dimension along which to perform the loess.
- **d** (*[0, 1]*) – Degree of the local regression.
- **f** (*float*) – Parameter controlling the shape of the weight curve. Behavior depends on the weighting function, but it usually represents the span of the weighting function in reference to x-coordinates normalized from 0 to 1.
- **niter** (*int*) – Number of robustness iterations to execute.
- **weights** (*["tricube", "gaussian"] or callable*) – Shape of the weighting function, see notes. The user can provide a function or a string: "tricube" : a smooth top-hat like curve. "gaussian" : a gaussian curve, f gives the span for 95% of the values.
- **equal_spacing** (*bool, optional*) – Whether to use the equal spacing optimization. If *None* (the default), it is activated only if the x-axis is equally-spaced. When activated, *dx = x[1] - x[0]*.
- **skipna** (*bool*) – If True (default), skip missing values (as marked by NaN). The output will have the same missing values as the input.

Notes

As stated in [Cleveland1979], the weighting function $W(x)$ should respect the following conditions:

- $W(x) > 0$ for $|x| < 1$
- $W(-x) = W(x)$
- $W(x)$ is non-increasing for $x \geq 0$
- $W(x) = 0$ for $|x| \geq 0$

If a Callable is provided, it should only accept the 1D *np.ndarray* x which is an absolute value function going from 1 to 0 to 1 around x_i, for all values where $x - x_i < h_i$ with h_i the distance of the rth nearest neighbor of x_i, $r = f * size(x)$.

References

Code adapted from https://gist.github.com/agramfort/850437

9.3.7 Properties Submodule

SDBA diagnostic tests are made up of statistical properties and measures. Properties are calculated on both simulation and reference datasets. They collapse the time dimension to one value.

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

`xclim.sdba.properties.STATISTICAL_PROPERTIES: dict[str, Callable] = {'acf': <function acf>, 'annual_cycle_amplitude': <function annual_cycle_amplitude>, 'annual_cycle_phase': <function annual_cycle_phase>, 'corr_btw_var': <function corr_btw_var>, 'mean': <function mean>, 'quantile': <function quantile>, 'relative_frequency': <function relative_frequency>, 'return_value': <function return_value>, 'skewness': <function skewness>, 'spell_length_distribution': <function spell_length_distribution>, 'trend': <function trend>, 'var': <function var>}`

Dictionary of all the statistical properties available.

`xclim.sdba.properties.acf`(*da: xr.DataArray*, ***, *lag: int = 1*, *group: str | Grouper = 'time.season'*) → xr.DataArray

Autocorrelation function.

Autocorrelation with a lag over a time resolution and averaged over all years.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **lag** (*int*) – Lag.
- **group** (*{'time.season', 'time.month'}*) – Grouping of the output. E.g. If 'time.month', the autocorrelation is calculated over each month separately for all years. Then, the autocorrelation for all Jan/Feb/... is averaged over all years, giving 12 outputs for each grid point.

Returns

xr.DataArray – lag-{lag} autocorrelation of the variable over a {group.prop} and averaged over all years.

See also:

statsmodels.tsa.stattools.acf

References

Alavoine M., and Grenier P. (under review) The distinct problems of physical inconsistency and of multivariate bias potentially involved in the statistical adjustment of climate simulations. International Journal of Climatology, submitted on September 19th 2021. (Preprint: https://doi.org/10.31223/X5C34C)

Examples

```
>>> from xclim.testing import open_dataset
>>> pr = open_dataset(path_to_pr_file).pr
>>> acf(da=pr, lag=3, group="time.season")
```

`xclim.sdba.properties.annual_cycle_amplitude(`*da: xr.DataArray*, ***, *amplitude_type: str = 'absolute'*, *group: str | Grouper = 'time'*`)` → xr.DataArray

Annual cycle amplitude.

The amplitudes of the annual cycle are calculated for each year, then averaged over the all years.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **amplitude_type** (*{'absolute','relative'}*) – Type of amplitude. 'absolute' is the peak-to-peak amplitude. (max - min). 'relative' is a relative percentage. 100 * (max - min) / mean (Recommended for precipitation).

Returns

xr.DataArray – {amplitude_type} amplitude of the annual cycle.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> annual_cycle_amplitude(da=pr, amplitude_type="relative")
```

`xclim.sdba.properties.annual_cycle_phase(`*da: xr.DataArray*, ***, *group: str | Grouper = 'time'*`)` → xr.DataArray

Annual cycle phase.

The phases of the annual cycle are calculated for each year, then averaged over the all years.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **group** (*{"time", 'time.season', 'time.month'}*) – Grouping of the output. Default: "time".

Returns

xr.DataArray – Phase of the annual cycle. The position (day-of-year) of the maximal value.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> annual_cycle_phase(da=pr)
```

`xclim.sdba.properties.corr_btw_var(`*da1: xr.DataArray*, *da2: xr.DataArray*, ***, *corr_type: str = 'Spearman'*, *group: str | Grouper = 'time'*, *output: str = 'correlation'*`)` → xr.DataArray

Correlation between two variables.

Spearman or Pearson correlation coefficient between two variables at the time resolution.

Parameters

- **da1** (*xr.DataArray*) – First variable on which to calculate the diagnostic.
- **da2** (*xr.DataArray*) – Second variable on which to calculate the diagnostic.
- **corr_type** (*{'Pearson','Spearman'}*) – Type of correlation to calculate.
- **output** (*{'correlation', 'pvalue'}*) – Wheter to return the correlation coefficient or the p-value.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. Eg. For 'time.month', the correlation would be calculated on each month separately, but with all the years together.

Returns

xr.DataArray – {corr_type} correlation coefficient

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> tasmax = open_dataset("NRCANdaily/nrcan_canada_daily_tasmax_1990.nc").tasmax
>>> corr_btw_var(da1=pr, da2=tasmax, group="time.season")
```

`xclim.sdba.properties.mean(`*da: xr.DataArray*, ***, *group: str | Grouper = 'time'*`)` → xr.DataArray

Mean.

Mean over all years at the time resolution.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. E.g. If 'time.month', the temporal average is performed separately for each month.

Returns

xr.DataArray, – Mean of the variable.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> mean(da=pr, group="time.season")
```

`xclim.sdba.properties.quantile(`*da: xr.DataArray*, ***, *q: float = 0.98*, *group: str | Grouper = 'time'*`)` → xr.DataArray

Quantile.

Returns the quantile q of the distribution of the variable over all years at the time resolution.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **q** (*float*) – Quantile to be calculated. Should be between 0 and 1.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. E.g. If 'time.month', the quantile is computed separately for each month.

Returns

xr.DataArray – Quantile {q} of the variable.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> quantile(da=pr, q=0.9, group="time.season")
```

`xclim.sdba.properties.relative_frequency(`*da: xr.DataArray*, ***, *op: str = '>='*, *thresh: str = '1mm d-1'*, *group: str | Grouper = 'time'*`)` → xr.DataArray

Relative Frequency.

Relative Frequency of days with variable respecting a condition (defined by an operation and a threshold) at the time resolution. The relative freqency is the number of days that satisfy the condition divided by the total number of days.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **op** (*{">", "<", ">=", "<="}*) – Operation to verify the condition. The condition is variable {op} threshold.
- **thresh** (*str*) – Threshold on which to evaluate the condition.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping on the output. Eg. For 'time.month', the relative frequency would be calculated on each month, with all years included.

Returns

xr.DataArray – Relative frequency of the variable.

Examples

```
>>> tasmax = open_dataset(path_to_tasmax_file).tasmax
>>> relative_frequency(da=tasmax, op="<", thresh="0 degC", group="time.season")
```

`xclim.sdba.properties.return_value(`*da: xr.DataArray*, *, *period: int = 20*, *op: str = 'max'*, *method: str = 'ML'*, *group: str | Grouper = 'time'*`)` → xr.DataArray

Return value.

Return the value corresponding to a return period. On average, the return value will be exceeded (or not exceed for op='min') every return period (eg. 20 years). The return value is computed by first extracting the variable annual maxima/minima, fitting a statistical distribution to the maxima/minima, then estimating the percentile associated with the return period (eg. 95th percentile (1/20) for 20 years)

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **period** (*int*) – Return period. Number of years over which to check if the value is exceeded (or not for op='min').
- **op** (*{'max','min'}*) – Whether we are looking for a probability of exceedance ('max', right side of the distribution) or a probability of non-exceedance (min, left side of the distribution).
- **method** (*{"ML", "PWM"}*) – Fitting method, either maximum likelihood (ML) or probability weighted moments (PWM), also called L-Moments. The PWM method is usually more robust to outliers. However, it requires the lmoments3 libraryto be installed from the *develop* branch. `pip install git+https://github.com/OpenHydrology/lmoments3.git@develop#egg=lmoments3`
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. A distribution of the extremums is done for each group.

Returns

xr.DataArray – {period}-{group} {op} return level of the variable.

Examples

```
>>> tas = open_dataset(path_to_tas_file).tas
>>> return_value(da=tas, group="time.season")
```

`xclim.sdba.properties.skewness(`*da: xr.DataArray*, *, *group: str | Grouper = 'time'*`)` → xr.DataArray

Skewness.

Skewness of the distribution of the variable over all years at the time resolution.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. E.g. If 'time.month', the skewness is performed separately for each month.

Returns

xr.DataArray – Skewness of the variable.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> skewness(da=pr, group="time.season")
```

See also:

`scipy.stats.skew`

`xclim.sdba.properties.spell_length_distribution`(*da: xr.DataArray*, ***, *method: str = 'amount'*, *op: str = '>='*, *thresh: str | float = '1 mm d-1'*, *stat: str = 'mean'*, *group: str |* Grouper *= 'time'*) → xr.DataArray

Spell length distribution.

Statistic of spell length distribution when the variable respects a condition (defined by an operation, a method and
a threshold).

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **method** (*{'amount', 'quantile'}*) – Method to choose the threshold. 'amount': The threshold is directly the quantity in {thresh}. It needs to have the same units as {da}. 'quantile': The threshold is calculated as the quantile {thresh} of the distribution.
- **op** (*{">", "<", ">=", "<="}*) – Operation to verify the condition for a spell. The condition for a spell is variable {op} threshold.
- **thresh** (*str or float*) – Threshold on which to evaluate the condition to have a spell. Str with units if the method is "amount". Float of the quantile if the method is "quantile".
- **stat** (*{'mean','max','min'}*) – Statistics to apply to the resampled input at the {group} (e.g. 1-31 Jan 1980) and then over all years (e.g. Jan 1980-2010)
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. E.g. If 'time.month', the spell lengths are coputed separately for each month.

Returns

xr.DataArray – {stat} of spell length distribution when the variable is {op} the {method} {thresh}.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> spell_length_distribution(da=pr, op="<", thresh="1mm d-1", group="time.season")
```

`xclim.sdba.properties.trend`(*da: xr.DataArray*, ***, *group: str |* Grouper *= 'time'*, *output: str = 'slope'*) → xr.DataArray

Linear Trend.

The data is averaged over each time resolution and the interannual trend is returned.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.

- **output** (*{'slope', 'pvalue'}*) – Attributes of the linear regression to return. 'slope' is the slope of the regression line. 'pvalue' is for a hypothesis test whose null hypothesis is that the slope is zero, using Wald Test with t-distribution of the test statistic.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping on the output.

Returns
xr.DataArray – Trend of the variable.

See also:

`scipy.stats.linregress`, `numpy.polyfit`

Examples

```
>>> tas = open_dataset(path_to_tas_file).tas
>>> trend(da=tas, group="time.season")
```

`xclim.sdba.properties.var(`*da: xr.DataArray*, ***, *group: str | Grouper = 'time'*`)` → xr.DataArray

Variance.

Variance of the variable over all years at the time resolution.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. E.g. If 'time.month', the variance is performed separately for each month.

Returns
xr.DataArray – Variance of the variable.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> var(da=pr, group="time.season")
```

9.3.8 Measures Submodule

SDBA diagnostic tests are made up of properties and measures. Measures compare adjusted simulations to a reference, through statistical properties or directly. This framework for the diagnostic tests was inspired by the [VALUE] project.

`xclim.sdba.measures.annual_cycle_correlation(`*sim*, *ref*, *window: int = 15*`)`

Annual cycle correlation.

Pearson correlation coefficient between the smooth day-of-year averaged annual cycles of the simulation and the reference. In the smooth day-of-year averaged annual cycles, each day-of-year is averaged over all years and over a window of days around that day.

Parameters

- **sim** (*xr.DataArray*) – data from the simulation (a time-series for each grid-point)

- **ref** (*xr.DataArray*) – data from the reference (observations) (a time-series for each grid-point)
- **window** (*int*) – Size of window around each day of year around which to take the mean. E.g. If window=31, Jan 1st is averaged over from December 17th to January 16th.

Returns
xr.DataArray, – Annual cycle correlation between the simulation and the reference

`xclim.sdba.measures.bias(`*sim: DataArray*, *ref: DataArray*`)` → DataArray

Bias.

The bias is the simulation minus the reference.

Parameters

- **sim** (*xr.DataArray*) – data from the simulation (one value for each grid-point)
- **ref** (*xr.DataArray*) – data from the reference (observations) (one value for each grid-point)

Returns
xr.DataArray, – Bias between the simulation and the reference

`xclim.sdba.measures.circular_bias(`*sim: DataArray*, *ref: DataArray*`)` → DataArray

Circular bias.

Bias considering circular time series. E.g. The bias between doy 365 and doy 1 is 364, but the circular bias is -1.

Parameters

- **sim** (*xr.DataArray*) – data from the simulation (one value for each grid-point)
- **ref** (*xr.DataArray*) – data from the reference (observations) (one value for each grid-point)

Returns
xr.DataArray, – Circular bias between the simulation and the reference

`xclim.sdba.measures.mae(`*sim: DataArray*, *ref: DataArray*`)` → DataArray

Mean absolute error.

The mean absolute error on the time dimension between the simulation and the reference.

Parameters

- **sim** (*xr.DataArray*) – data from the simulation (a time-series for each grid-point)
- **ref** (*xr.DataArray*) – data from the reference (observations) (a time-series for each grid-point)

Returns
xr.DataArray, – Mean absolute error between the simulation and the reference

`xclim.sdba.measures.ratio(`*sim: DataArray*, *ref: DataArray*`)` → DataArray

Ratio.

The ratio is the quotient of the simulation over the reference.

Parameters

- **sim** (*xr.DataArray*) – data from the simulation (one value for each grid-point)

- **ref** (*xr.DataArray*) – data from the reference (observations) (one value for each grid-point)

Returns
xr.DataArray, – Ratio between the simulation and the reference

`xclim.sdba.measures.relative_bias(`*sim: DataArray*, *ref: DataArray*`)` → DataArray

Relative Bias.

The relative bias is the simulation minus reference, divided by the reference.

Parameters

- **sim** (*xr.DataArray*) – data from the simulation (one value for each grid-point)
- **ref** (*xr.DataArray*) – data from the reference (observations) (one value for each grid-point)

Returns
xr.DataArray, – Relative bias between the simulation and the reference

`xclim.sdba.measures.rmse(`*sim: DataArray*, *ref: DataArray*`)` → DataArray

Root mean square error.

The root mean square error on the time dimension between the simulation and the reference.

Parameters

- **sim** (*xr.DataArray*) – Data from the simulation (a time-series for each grid-point)
- **ref** (*xr.DataArray*) – Data from the reference (observations) (a time-series for each grid-point)

Returns
xr.DataArray, – Root mean square error between the simulation and the reference

9.4 Developer tools

9.4.1 Base Classes and Developer Tools

`class xclim.sdba.base.Parametrizable`

Bases: `dict`

Helper base class resembling a dictionary.

This object is _completely_ defined by the content of its internal dictionary, accessible through item access (*self['attr']*) or in *self.parameters*. When serializing and restoring this object, only members of that internal dict are preserved. All other attributes set directly with *self.attr = value* will not be preserved upon serialization and restoration of the object with *[json]pickle*. dictionary. Other variables set with *self.var = data* will be lost in the serialization process. This class is best serialized and restored with *jsonpickle*.

`property parameters`

All parameters as a dictionary. Read-only.

`class xclim.sdba.base.ParametrizableWithDataset`

Bases: *`Parametrizable`*

Parametrizeable class that also has a *ds* attribute storing a dataset.

classmethod `from_dataset`(*ds: Dataset*)

Create an instance from a dataset.

The dataset must have a global attribute with a name corresponding to *cls._attribute*, and that attribute must be the result of *jsonpickle.encode(object)* where object is of the same type as this object.

`set_dataset`(*ds: Dataset*)

Store an xarray dataset in the *ds* attribute.

Useful with custom object initialization or if some external processing was performed.

`xclim.sdba.base.duck_empty`(*dims*, *sizes*, *dtype='float64'*, *chunks=None*)

Return an empty DataArray based on a numpy or dask backend, depending on the chunks argument.

`xclim.sdba.base.map_blocks`(*reduces: Optional[Sequence[str]] = None*, ***outvars*)

Decorator for declaring functions and wrapping them into a map_blocks.

Takes care of constructing the template dataset. Dimension order is not preserved. The decorated function must always have the signature: `func(ds, **kwargs)`, where ds is a DataArray or a Dataset. It must always output a dataset matching the mapping passed to the decorator.

Parameters

- **reduces** (*sequence of strings*) – Name of the dimensions that are removed by the function.
- **outvars** – Mapping from variable names in the output to their *new* dimensions. The placeholders *Grouper.PROP*, *Grouper.DIM* and *Grouper.ADD_DIMS* can be used to signify *group.prop*,`group.dim` and *group.add_dims* respectively. If an output keeps a dimension that another loses, that dimension name must be given in *reduces* and in the list of new dimensions of the first output.

`xclim.sdba.base.map_groups`(*reduces: Optional[Sequence[str]] = None*, *main_only: bool = False*, ***out_vars*)

Decorator for declaring functions acting only on groups and wrapping them into a map_blocks.

This is the same as *map_blocks* but adds a call to *group.apply()* in the mapped func and the default value of *reduces* is changed.

The decorated function must have the signature: `func(ds, dim, **kwargs)`. Where ds is a DataAray or Dataset, dim is the group.dim (and add_dims). The *group* argument is stripped from the kwargs, but must evidently be provided in the call.

Parameters

- **reduces** (*sequence of str*) – Dimensions that are removed from the inputs by the function. Defaults to [Grouper.DIM, Grouper.ADD_DIMS] if main_only is False, and [Grouper.DIM] if main_only is True. See `map_blocks()`.
- **main_only** (*bool*) – Same as for `Grouper.apply()`.
- **out_vars** – Mapping from variable names in the output to their *new* dimensions. The placeholders *Grouper.PROP*, *Grouper.DIM* and *Grouper.ADD_DIMS* can be used to signify *group.prop*,`group.dim` and *group.add_dims* respectively. If an output keeps a dimension that another loses, that dimension name must be given in *reduces* and in the list of new dimensions of the first output.

See also:

`map_blocks()`

`xclim.sdba.base.parse_group`(*func: Callable*, *kwargs=None*, *allow_only=None*) → Callable

Parse the kwargs given to a function to set the *group* arg with a Grouper object.

This function can be used as a decorator, in which case the parsing and updating of the kwargs is done at call time. It can also be called with a function from which extract the default group and kwargs to update, in which case it returns the updated kwargs.

If allow_only is given, an exception is raised when the parsed group is not within that list.

class `xclim.sdba.detrending.BaseDetrend`(**, *group: Grouper | str = 'time'*, *kind: str = '+'*, ***kwargs*)

Base class for detrending objects.

Defines three methods:

fit(da) : Compute trend from da and return a new _fitted_ Detrend object. detrend(da) : Return detrended array. retrend(da) : Puts trend back on da.

A fitted Detrend object is unique to the trend coordinate of the object used in *fit*, (usually 'time'). The computed trend is stored in `Detrend.ds.trend`.

Subclasses should implement `_get_trend_group()` or `_get_trend()`. The first will be called in a `group.apply(..., main_only=True)`, and should return a single DataArray. The second allows the use of functions wrapped in `map_groups()` and should also return a single DataArray.

The subclasses may reimplement `_detrend` and `_retrend`.

`detrend`(*da: DataArray*)

Remove the previously fitted trend from a DataArray.

`fit`(*da: DataArray*)

Extract the trend of a DataArray along a specific dimension.

Returns a new object that can be used for detrending and retrending. Fitted objects are unique to the fitted coordinate used.

property `fitted`

Return whether instance is fitted.

`retrend`(*da: DataArray*)

Put the previously fitted trend back on a DataArray.

class `xclim.sdba.adjustment.TrainAdjust`(**args*, *_trained=False*, ***kwargs*)

Base class for adjustment objects obeying the train-adjust scheme.

Children classes should implement these methods:

- `_train(ref, hist, **kwargs)`, classmethod receiving the training target and data, returning a training dataset and parameters to store in the object.
- `_adjust(sim, **kwargs)`, receiving the projected data and some arguments, returning the *scen* dataarray.

`adjust`(*sim: DataArray*, **args*, ***kwargs*)

Return bias-adjusted data. Refer to the class documentation for the algorithm details.

Parameters

- **sim** (*DataArray*) – Time series to be bias-adjusted, usually a model output.
- **args** (*xr.DataArray*) – Other DataArrays needed for the adjustment (usually none).
- **kwargs** – Algorithm-specific keyword arguments, see class doc.

`set_dataset(`*ds: Dataset*`)`

Store an xarray dataset in the *ds* attribute.

Useful with custom object initialization or if some external processing was performed.

`classmethod train(`*ref: DataArray*, *hist: DataArray*, ***kwargs*`)`

Train the adjustment object. Refer to the class documentation for the algorithm details.

Parameters

- **ref** (*DataArray*) – Training target, usually a reference time series drawn from observations.
- **hist** (*DataArray*) – Training data, usually a model output whose biases are to be adjusted.

`class xclim.sdba.adjustment.Adjust(`**args*, *_trained=False*, ***kwargs*`)`

Adjustment with no intermediate trained object.

Children classes should implement a *_adjust* classmethod taking as input the three DataArrays and returning the scen dataset/array.

`classmethod adjust(`*ref: DataArray*, *hist: DataArray*, *sim: DataArray*, ***kwargs*`)`

Return bias-adjusted data. Refer to the class documentation for the algorithm details.

Parameters

- **ref** (*DataArray*) – Training target, usually a reference time series drawn from observations.
- **hist** (*DataArray*) – Training data, usually a model output whose biases are to be adjusted.
- **sim** (*DataArray*) – Time series to be bias-adjusted, usually a model output.
- **kwargs** – Algorithm-specific keyword arguments, see class doc.

`xclim.sdba.properties.register_statistical_properties(`*aspect: str*, *seasonal: bool*, *annual: bool*`)` → Callable

Register statistical properties in the STATISTICAL_PROPERTIES dictionary with its aspect and time resolutions.

`xclim.sdba.measures.check_same_units_and_convert(`*func*`)` → Callable

Verify that the simulation and the reference have the same units.

If not, it converts the simulation to the units of the reference.

CHAPTER

TEN

SPATIAL ANALOGUES

Spatial analogues are maps showing which areas have a present-day climate that is analogous to the future climate of a given place. This type of map can be useful for climate adaptation to see how well regions are coping today under specific climate conditions. For example, officials from a city located in a temperate region that may be expecting more heatwaves in the future can learn from the experience of another city where heatwaves are a common occurrence, leading to more proactive intervention plans to better deal with new climate conditions.

Spatial analogues are estimated by comparing the distribution of climate indices computed at the target location over the future period with the distribution of the same climate indices computed over a reference period for multiple candidate regions. A number of methodological choices thus enter the computation:

- Climate indices of interest,
- Metrics measuring the difference between both distributions,
- Reference data from which to compute the base indices,
- A future climate scenario to compute the target indices.

The climate indices chosen to compute the spatial analogues are usually annual values of indices relevant to the intended audience of these maps. For example, in the case of the wine grape industry, the climate indices examined could include the length of the frost-free season, growing degree-days, annual winter minimum temperature and annual number of very cold days [Roy2017].

See *Spatial Analogues examples*.

10.1 Methods to compute the (dis)similarity between samples

This module implements all methods described in [Grenier2013] to measure the dissimilarity between two samples, plus the Székely-Rizzo energy distance, Some of these algorithms can be used to test whether two samples have been drawn from the same distribution. Here, they are used in finding areas with analogue climate conditions to a target climate.

- Standardized Euclidean distance
- Nearest Neighbour distance
- Zech-Aslan energy statistic
- Székely-Rizzo energy distance
- Friedman-Rafsky runs statistic
- Kolmogorov-Smirnov statistic
- Kullback-Leibler divergence

All methods accept arrays, the first is the reference (n, D) and the second is the candidate (m, D). Where the climate indicators vary along D and the distribution dimension along n or m. All methods output a single float. See their documentation in *Analogue metrics API*.

> **Warning:** Some methods are scale-invariant and others are not. This is indicated in the docstring of the methods as it can change the results significantly. In most cases, scale-invariance is desirable and inputs may need to be scaled beforehand for scale-dependent methods.

References

`xclim.analog.spatial_analogs(`*target: xr.Dataset*, *candidates: xr.Dataset*, *dist_dim: str | Sequence[str] = 'time'*, *method: str = 'kldiv'*, ***kwargs*`)`

Compute dissimilarity statistics between target points and candidate points.

Spatial analogues based on the comparison of climate indices. The algorithm compares the distribution of the reference indices with the distribution of spatially distributed candidate indices and returns a value measuring the dissimilarity between both distributions over the candidate grid.

Parameters

- **target** (*xr.Dataset*) – Dataset of the target indices. Only indice variables should be included in the dataset's *data_vars*. They should have only the dimension(s) *dist_dim `in common with `candidates*.
- **candidates** (*xr.Dataset*) – Dataset of the candidate indices. Only indice variables should be included in the dataset's *data_vars*.
- **dist_dim** (*str*) – The dimension over which the *distributions* are constructed. This can be a multi-index dimension.
- **method** (*{'seuclidean', 'nearest_neighbor', 'zech_aslan', 'kolmogorov_smirnov', 'friedman_rafsky', 'kldiv'}*) – Which method to use when computing the dissimilarity statistic.
- **kwargs** – Any other parameter passed directly to the dissimilarity method.

Returns

xr.DataArray – The dissimilarity statistic over the union of candidates' and target's dimensions. The range depends on the method.

10.2 Analogue metrics API

`xclim.analog.friedman_rafsky(`*x: ndarray*, *y: ndarray*`)` → float

Compute a dissimilarity metric based on the Friedman-Rafsky runs statistics.

The algorithm builds a minimal spanning tree (the subset of edges connecting all points that minimizes the total edge length) then counts the edges linking points from the same distribution. This method is scale-dependent.

Parameters

- **x** (*np.ndarray (n,d)*) – Reference sample.
- **y** (*np.ndarray (m,d)*) – Candidate sample.

Returns

float – Friedman-Rafsky dissimilarity metric ranging from 0 to (m+n-1)/(m+n).

References

Friedman J.H. and Rafsky, L.C. (1979) Multivariate generalisations of the Wald-Wolfowitz and Smirnov two-sample tests. Annals of Stat. Vol.7, No. 4, 697-717. https://doi.org/10.1214/aos/1176344722.

`xclim.analog.kldiv(`*x: np.ndarray, y: np.ndarray, *, k: int | Sequence[int] = 1*`)` → float | Sequence[float]

Compute the Kullback-Leibler divergence between two multivariate samples.

where $r_k(x_i)$ and $s_k(x_i)$ are, respectively, the euclidean distance to the kth neighbour of x_i in the x array (excepting x_i) and in the y array. This method is scale-dependent.

Parameters

- **x** (*np.ndarray (n,d)*) – Samples from distribution P, which typically represents the true distribution (reference).
- **y** (*np.ndarray (m,d)*) – Samples from distribution Q, which typically represents the approximate distribution (candidate)
- **k** (*int or sequence*) – The kth neighbours to look for when estimating the density of the distributions. Defaults to 1, which can be noisy.

Returns

float or sequence – The estimated Kullback-Leibler divergence D(P||Q) computed from the distances to the kth neighbour.

Notes

In information theory, the Kullback–Leibler divergence ([perezcruz08]) is a non-symmetric measure of the difference between two probability distributions P and Q, where P is the "true" distribution and Q an approximation. This nuance is important because $D(P||Q)$ is not equal to $D(Q||P)$.

For probability distributions P and Q of a continuous random variable, the K–L divergence is defined as:

$$D_{KL}(P||Q) = \int p(x) \log \left(\frac{p(x)}{q(x)} \right) dx$$

This formula assumes we have a representation of the probability densities $p(x)$ and $q(x)$. In many cases, we only have samples from the distribution, and most methods first estimate the densities from the samples and then proceed to compute the K-L divergence. In Perez-Cruz, the authors propose an algorithm to estimate the K-L divergence directly from the sample using an empirical CDF. Even though the CDFs do not converge to their true values, the paper proves that the K-L divergence almost surely does converge to its true value.

References

`xclim.analog.kolmogorov_smirnov(`*x: ndarray*, *y: ndarray*`)` → float

Compute the Kolmogorov-Smirnov statistic applied to two multivariate samples as described by Fasano and Franceschini.

This method is scale-dependent.

Parameters

- **x** (*np.ndarray (n,d)*) – Reference sample.
- **y** (*np.ndarray (m,d)*) – Candidate sample.

Returns
float – Kolmogorov-Smirnov dissimilarity metric ranging from 0 to 1.

References

Fasano, G., & Franceschini, A. (1987). A multidimensional version of the Kolmogorov-Smirnov test. Monthly Notices of the Royal Astronomical Society, 225, 155-170. https://doi.org/10.1093/mnras/225.1.155

`xclim.analog.nearest_neighbor(`*x: ndarray*, *y: ndarray*`)` → ndarray

Compute a dissimilarity metric based on the number of points in the pooled sample whose nearest neighbor belongs to the same distribution.

This method is scale-invariant.

Parameters

- **x** (*np.ndarray (n,d)*) – Reference sample.
- **y** (*np.ndarray (m,d)*) – Candidate sample.

Returns
float – Nearest-Neighbor dissimilarity metric ranging from 0 to 1.

References

Henze N. (1988) A Multivariate two-sample test based on the number of nearest neighbor type coincidences. Ann. of Stat., Vol. 16, No.2, 772-783. https://doi.org/10.1214/aos/1176350835.

`xclim.analog.seuclidean(`*x: ndarray*, *y: ndarray*`)` → float

Compute the Euclidean distance between the mean of a multivariate candidate sample with respect to the mean of a reference sample.

This method is scale-invariant.

Parameters

- **x** (*np.ndarray (n,d)*) – Reference sample.
- **y** (*np.ndarray (m,d)*) – Candidate sample.

Returns
float – Standardized Euclidean Distance between the mean of the samples ranging from 0 to infinity.

Notes

This metric considers neither the information from individual points nor the standard deviation of the candidate distribution.

References

Veloz et al. (2011) Identifying climatic analogs for Wisconsin under 21st-century climate-change scenarios. Climatic Change, https://doi.org/10.1007/s10584-011-0261-z.

`xclim.analog.szekely_rizzo(`*x: ndarray*, *y: ndarray*, ***, *standardize: bool = True*`)` → float

Compute the Székely-Rizzo energy distance dissimilarity metric based on an analogy with Newton's gravitational potential energy.

This method is scale-invariant when *standardize=True* (default), scale-dependent otherwise.

Parameters

- **x** (*ndarray (n,d)*) – Reference sample.
- **y** (*ndarray (m,d)*) – Candidate sample.
- **standardize** (*bool*) – If True (default), the standardized euclidean norm is used, instead of the conventional one.

Returns

float – Székely-Rizzo's energy distance dissimilarity metric ranging from 0 to infinity.

Notes

The e-distance between two variables X, Y (target and candidates) of sizes n, d and m, d proposed by [SR2004] is defined by:

$$e(X,Y) = \frac{nm}{n+m}\left[2\phi_{xy}\phi_{xx}\phi_{yy}\right]$$

where

$$\phi_{xy} = \frac{1}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m}\|X_iY_j\|$$

$$\phi_{xx} = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\|X_iX_j\|$$

$$\phi_{yy} = \frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m}\|X_iY_j\|$$

and where $\|\cdot\|$ denotes the Euclidean norm, X_i denotes the i-th observation of X. When *standardized=False*, this corresponds to the T test of [RS2016] (p. 28) and to the `eqdist.e` function of the *energy* R package (with two samples) and gives results twice as big as *`xclim.sdba.processing.escore()`*. The standardization was added following the logic of [Grenier2013] to make the metric scale-invariant.

References

`xclim.analog.zech_aslan(x: ndarray, y: ndarray, *, dmin: float = 1e-12) → float`

Compute a modified Zech-Aslan energy distance dissimilarity metric based on an analogy with the energy of a cloud of electrical charges.

This method is scale-invariant.

Parameters

- **x** (*np.ndarray (n,d)*) – Reference sample.
- **y** (*np.ndarray (m,d)*) – Candidate sample.
- **dmin** (*float*) – The cut-off for low distances to avoid singularities on identical points.

Returns

float – Zech-Aslan dissimilarity metric ranging from -infinity to infinity.

Notes

The energy measure between two variables X, Y (target and candidates) of sizes n, d and m, d proposed by [AZ03] is defined by:

$$\begin{aligned}
e(X,Y) &= [\phi_{xx} + \phi_{yy} - \phi_{xy}] \\
\phi_{xy} &= \frac{1}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m} R\left[SED(X_i, Y_j)\right] \\
\phi_{xx} &= \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=i+1}^{n} R\left[SED(X_i, X_j)\right] \\
\phi_{yy} &= \frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=i+1}^{m} R\left[SED(X_i, Y_j)\right]
\end{aligned}$$

where X_i denotes the i-th observation of X. R is a weight function and $SED(A, B)$ denotes the standardized Euclidean distance.

$$\begin{aligned}
R(r) &= \begin{cases} -\ln r & \text{for } r > d_{min} \\ -\ln d_{min} & \text{for } r \leq d_{min} \end{cases} \\
SED(X_i, Y_j) &= \sqrt{\sum_{k=1}^{d} \frac{(X_i(k) - Y_i(k))^2}{\sigma_x(k)\sigma_y(k)}}
\end{aligned}$$

where k is a counter over dimensions (indices in the case of spatial analogs) and $\sigma_x(k)$ is the standard deviation of X in dimension k. Finally, d_{min} is a cut-off to avoid poles when $r \to 0$, it is controllable through the *dmin* parameter.

This version corresponds the D_{ZAE} test of [Grenier2013] (eq. 7), which is a version of ϕ_{NM} from [AZ03], modified by using the standardized euclidean distance, the log weight function and choosing $d_{min} = 10^{-12}$.

References

10.3 Utilities for developers

`xclim.analog.metric(`*func*`)`

Register a metric function in the *metrics* mapping and add some preparation/checking code.

All metric functions accept 2D inputs. This reshapes 1D inputs to (n, 1) and (m, 1). All metric functions are invalid when any non-finite values are present in the inputs.

`xclim.analog.standardize(`*x: np.ndarray*, *y: np.ndarray*`)` → tuple[np.ndarray, np.ndarray]

Standardize x and y by the square root of the product of their standard deviation.

Parameters

- **x** (*np.ndarray*) – Array to be compared.
- **y** (*np.ndarray*) – Array to be compared.

Returns

(ndarray, ndarray) – Standardized arrays.

CHAPTER

ELEVEN

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:

11.1 Types of Contributions

11.1.1 Implement Features, Indices or Indicators

xclim's structure makes it easy to create and register new user-defined indices and indicators. For the general implementation of indices and their wrapping into indicators, refer to *Extending xclim* and *Customizing and controlling xclim*.

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

General to-do list for implementing a new Indicator:

1. Implement the indice
   - Indices are function wrapped with `declare_units()`
   - Their input arguments should have type annotations, as documented in `InputKind`
   - Their docstring should follow the scheme explained in *Defining new indices*.
   - They should set the units on their outputs, but no other metadata fields.
   - Their code should be found in the most relevant `xclim/indices/_*.py` file. Functions are explicitly added to the `__all__` at the top of the file.
2. Add unit tests
   - Indices are best tested with made up, idealized data to explicitly test the edge cases. Many pytest fixtures are available to help this data generation.
   - Tests should be added as one or more functions in `xclim/testing/tests/test_indices.py`, see other tests for inspiration.
3. Add the indicator
   - See *Defining new indicators* for more info and look at the other indicators for inspiration.
   - They are added in the most relevant `xclim/indicators/{realm}/_*.py` file.

   - Indicator are instances of subclasses of *xclim.core.indicator.Indicator*. They should use a class declared within the `{realm}` folder, creating a dummy one if needed. They are explicitly added to the file's `__all__`.

4. Add unit tests

   - Indicators are best tested with real data, also looking at missing value propagation and metadata formatting. In addition to the `atmos_ds` fixture, only datasets that can be accessed with `xclim.testing.open_dataset()` should be used.
   - Tests are added in the most relevant `xclim/testing/tests/test_{variable}.py` file.

5. Add french translations

   xclim comes with an internationalization module and all "official" indicators (those in `xclim.atmos.indicators`) must have a french translation added to `xclim/data/fr.json`. This part can be done by the core team after you open a PR.

General notes for implementing new bias-adjustment methods:

- Method are implemented as classes in `xclim/sdba/adjustment.py`.
- If the algorithm gets complicated and would generate many dask tasks, it should be implemented as functions wrapped by `map_blocks()` or `map_groups()` in `xclim/sdba/_adjustment.py`.
- xclim doesn't implement monolithic multi-parameter methods, but rather smaller modular functions to construct post-processing workflows.

11.1.2 Report Bugs

Report bugs at https://github.com/Ouranosinc/xclim/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.

11.1.3 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

11.1.4 Write Documentation

xclim could always use more documentation, whether as part of the official xclim docs, in docstrings, or even on the web in blog posts, articles, and such.

11.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/Ouranosinc/xclim/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.
- The Xclim development team welcomes you and is always on hand to help. :)

11.2 Get Started!

Ready to contribute? Here's how to set up *xclim* for local development.

1. Fork the *xclim* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:{my_github_username}/xclim.git
   $ cd xclim/
   ```

3. Create a development environment. We recommend using `conda`:

   ```
   $ conda create -n xclim python=3.8 --file=environment.yml
   $ pip install -e .[dev]
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally!

5. Before committing your changes, we ask that you install `pre-commit` in your development environment. Pre-commit runs git hooks that ensure that your code resembles that of the project and catches and corrects any small errors or inconsistencies when you `git commit`:

   ```
   # To install the necessary pre-commit hooks:
   $ pre-commit install
   # To run pre-commit hooks manually:
   $ pre-commit run --all-files
   ```

   Instead of `pre-commit`, you could also verify your changes manually with *black*, *flake8*, *flake8-rst-docstrings*, *pydocstyle*, and *yamllint*:

   ```
   $ black --check --target-version py38 xclim xclim/testing/tests
   $ black --check --target-version py38 --include "\.ipynb$" docs
   $ flake8 xclim xclim/testing/tests
   $ pydocstyle --config=setup.cfg xclim xclim
   $ yamllint --config-file .yamllint.yaml xclim
   ```

6. When unit/doc tests are added or notebooks updated, use `pytest` to run them. Alternatively, one can use `tox` to run all testing suites as would github do when the PR is submitted and new commits are pushed:

```
$ pytest --nbval docs/notebooks  # for notebooks, exclusively.
$ pytest --rootdir xclim/testing/tests/ --xdoctest xclim --ignore=xclim/testing/
↪tests/  # for doctests, exclusively.
$ pytest  # for all unit tests, excluding doctests and notebooks.
$ tox  # run all testing suites
```

7. Docs should also be tested to ensure that the documentation will build correctly on ReadTheDocs. This can be performed in a number of ways:

```
# To run in a contained virtualenv environment
$ tox -e docs
# or, alternatively, to build the docs directly
$ make docs
```

8. After clearing the previous checks, commit your changes and push your branch to GitHub:

```
$ git add *

$ git commit -m "Your detailed description of your changes."
```

If installed, *pre-commit* will run checks at this point:

- If no errors are found, changes will be committed.
- If errors are found, modifications will be made and warnings will be raised if intervention is needed.
- After adding changes, simply *git commit* again:

```
$ git push origin name-of-your-bugfix-or-feature
```

9. Submit a pull request through the GitHub website.

11.3 Pull Request Guidelines

Before you submit a pull request, please follow these guidelines:

1. Open an *issue* on our GitHub repository with your issue that you'd like to fix or feature that you'd like to implement.
2. Perform the changes, commit and push them either to new a branch within Ouranosinc/xclim or to your personal fork of xclim.

Warning: Try to keep your contributions within the scope of the issue that you are addressing. While it might be tempting to fix other aspects of the library as it comes up, it's better to simply to flag the problems in case others are already working on it.

Consider adding a "**# TODO:**" comment if the need arises.

3. Pull requests should raise test coverage for the xclim library. Code coverage is an indicator of how extensively tested the library is. If you are adding a new set of functions, they **must be tested** and **coverage percentage should not significantly decrease.**
4. If the pull request adds functionality, your functions should include docstring explanations. So long as the docstrings are syntactically correct, sphinx-autodoc will be able to automatically parse the

information. Please ensure that the docstrings and documentation adhere to the following standards (badly formed docstrings will fail build tests):

- numpydoc
- reStructuredText (ReST)

Note: If you aren't accustomed to writing documentation in reStructuredText (*.rst*), we encourage you to spend a few minutes going over the incredibly well-summarized reStructuredText Primer from the sphinx-doc maintainer community.

5. The pull request should work for Python 3.8, 3.9, and 3.10 as well as raise test coverage. Pull requests are also checked for documentation build status and for PEP8 compliance.

   The build statuses and build errors for pull requests can be found at:
   https://github.com/Ouranosinc/xclim/actions

Warning: PEP8, black, pytest (with xdoctest) and pydocstyle (for numpy docstrings) conventions are strongly enforced. Ensure that your changes pass all tests prior to pushing your final commits to your branch. Code formatting errors are treated as build errors and will block your pull request from being accepted.

6. The version changes (HISTORY.rst) should briefly describe changes introduced in the Pull request. Changes should be organized by type (ie: *New indicators*, *New features and enhancements*, *Breaking changes*, *Bug fixes*, *Internal changes*) and the GitHub Pull Request, GitHub Issue. Your name and/or GitHub handle should also be listed among the contributors to this version. This can be done as follows:

   ```
   Contributors to this version: John Jacob Jingleheimer Schmidt (:user:`username`).

   Internal changes
   ^^^^^^^^^^^^^^^^
   * Updated the contribution guidelines. (:issue:`868`, :pull:`869`).
   ```

   If this is your first contribution to Ouranosinc/xclim, we ask that you also add your name to the AUTHORS.rst, under *Contributors*.

11.4 Tips

To run a subset of tests, we suggest a few approaches. For running only a test file:

```
$ pytest xclim/testing/tests/test_xclim.py
```

To skip all slow tests:

```
$ pytest -m "not slow"
```

To run all conventions tests at once:

```
$ pre-commit run --all-files
```

11.5 Versioning

In order to update and release the library to PyPI, it's good to use a semantic versioning scheme. The method we use is as follows:

```
major.minor.patch-release
```

Major releases denote major changes resulting in a stable API;

Minor is to be used when adding a module, process or set of components;

Patch should be used for bug fixes and optimizations;

Release is a keyword used to specify the degree of production readiness (*beta* [, and optionally, *gamma*]). *Only versions built from the main development branch will ever have this tag!*

> An increment to the Major or Minor will reset the Release to *beta*. When a build is promoted above *beta* (ie: release-ready), it's a good idea to push this version towards PyPi.

11.6 Deploying

A reminder for the maintainers on how to prepare the library for a tagged version.

Make sure all your changes are committed (**including an entry in HISTORY.rst**). Then run:

```
$ bump2version <option>  # possible options: major / minor / patch / release
```

These commands will increment the version and create a commit with an autogenerated message.

For PyPI releases/stable versions, ensure that the last version bumping command run is *$ bump2version release* to remove the *-dev*. These changes can now be merged to the main development branch:

```
$ git push
```

With this performed, we can tag a version that will act as the GitHub-provided stable source archive. Be sure to only tag from the *main* branch when all changes from PRs have been merged! Commands needed are:

```
$ git tag v1.2.3-XYZ
$ git push --tags
```

Note: Starting from October, 2021, all tags pushed to GitHub will trigger a build and publish a package to TestPyPI by default. TestPyPI is a testing ground that is not indexed or easily available to *pip*. The test package can be found at: https://test.pypi.org/project/xclim/

11.7 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.

11.7.1 The Automated Approach

The simplest way to package *xclim* is to "publish" a version on GitHuh. GitHub CI Actions are presently configured to build the library and publish the packages on PyPI automatically.

When publishing on GitHub, maintainers will need to generate the release notes for the current version, replacing the `:issue:`, `:pull:`, and `:user:` tags. The *xclim* CLI offers a helper function for performing this action:

```
# For Markdown format (needed when publishing a new version on GitHub):
$ xclim release_notes -m
# For ReStructuredText format (offered for convenience):
$ xclim release_notes -r
```

When publishing to GitHub, you will still need to replace subsection headers in the Markdown (`^^^^ -> ###`) and the history published should not extend past the changes for the current version. This behaviour may eventually change.

Warning: Be warned that a published package version on PyPI can never be overwritten. Be sure to verify that the package published at https://test.pypi.org/project/xclim/ matches expectations before publishing a version on GitHub.

11.7.2 The Manual Approach

The manual approach to library packaging for general support (pip wheels) requires the following packages installed:

- setuptools
- wheel
- twine

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 setup.py sdist bdist_wheel

# To upload to PyPI
$ twine upload dist/*
```

The new version based off of the version checked out will now be available via *pip* (*$ pip install xclim*).

11.7.3 Releasing on conda-forge

Initial Release

In order to prepare an initial release on conda-forge, we *strongly* suggest consulting the following links:

- https://conda-forge.org/docs/maintainer/adding_pkgs.html
- https://github.com/conda-forge/staged-recipes

Subsequent releases

If the conda-forge feedstock recipe is built from PyPI, then when a new release is published on PyPI, *regro-cf-autotick-bot* will open Pull Requests automatically on the conda-forge feedstock. It is up to the conda-forge feedstock maintainers to verify that the package is building properly before merging the Pull Request to the main branch.

Before updating the main conda-forge recipe, we *strongly* suggest performing the following checks:

- Ensure that dependencies and dependency versions correspond with those of the tagged version, with open or pinned versions for the *host* requirements.
- If possible, configure tests within the conda-forge build CI (e.g. *imports: xclim*, *commands: pytest xclim*)

CHAPTER

TWELVE

CREDITS

12.1 Development Lead

- Travis Logan <logan.travis@ouranos.ca> @tlogan2000

12.2 Co-Developers

- Abel Aoun <aoun.abel@gmail.com> @bzah
- Pascal Bourgault <bourgault.pascal@ouranos.ca> @aulemahal
- David Huard <huard.david@ouranos.ca> @huard
- Juliette Lavoie <lavoie.juliette@ouranos.ca> @juliettelavoie
- Gabriel Rondeau-Genesse <rondeau-genesse.gabriel@ouranos.ca> @RondeauG
- Trevor James Smith <smith.trevorj@ouranos.ca> @Zeitsperre

12.3 Contributors

- Raquel Alegre <raquel.alegre@gmail.com> @raquel-ucl
- Clair Barnes <clair.barnes.16@ucl.ac.uk> @clairbarnes
- Sébastien Biner <biner.sebastien@ouranos.ca> @sbiner
- David Caron @davidcaron
- Carsten Ehbrecht <ehbrecht@dkrz.de> @cehbrecht
- Jeremy Fyke @jeremyfyke
- Tom Keel <thomas.keel.18@ucl.ac.uk> @Thomasjkeel
- Jwen Fai Low @jwenfai
- Marie-Pier Labonté @marielabonte
- Jamie Quinn <jamiejquinn@jamiejquinn.com> @JamieJQuinn
- Philippe Roy <roy.philippe@ouranos.ca> @Balinus
- Yannick Rousseau @yrouranos
- Ag Stephens <ag.stephens@stfc.ac.uk> @agstephens

- Maliko Tanguy <malngu@ceh.ac.uk> @malngu

CHAPTER

THIRTEEN

HISTORY

13.1 0.37.0 (17 June 2022)

Contributors to this version: Abel Aoun (@bzah), Pascal Bourgault (@aulemahal), Trevor James Smith (@Zeitsperre), Gabriel Rondeau-Genesse (@RondeauG), Juliette Lavoie (@juliettelavoie), Ludwig Lierhammer (@ludwiglierhammer).

13.1.1 Announcements

- *xclim* is now compliant with PEP 563. Python3.10-style annotations are now permitted. (GH/1065, PR/1071).
- *xclim* is now fully compatible with *xarray*'s *flox*-enabled `GroupBy` and `resample` operations. (PR/1081).
- *xclim* now (properly) enforces docstring compliance checks using *pydocstyle* with modified *numpy*-style docstrings. Docstring errors will now cause build failures. See the pydocstyle documentation for more information. (PR/1074).
- *xclim* now uses GitHub Actions to manage patch version bumping. Merged Pull Requests that modify *xclim* code now trigger version-bumping automatically when pushed to the main development branch. Running *$ bump2version patch* within development branches is no longer necessary. (PR/1102).

13.1.2 New features and enhancements

- Add "Celsius" to aliases of "celsius" unit. (GH/1067, PR/1068).
- All indicators now have indexing enabled, except those computing statistics on spells. (GH/1069, PR/1070).
- **A convenience function for returning the version numbers for relevant xclim dependencies (`xclim.testing.show_versions`) is now offered. (PR/1073).**
  - A CLI version of this function is also available from the command line (*$ xclim show_version_info*). (PR/1073).
- New "keep_attrs" option to control the handling of the attributes within the indicators. (GH/1026, PR/1076).
- Added a notebook showcasing some simple examples of Spatial Analogues. (GH/585, PR/1075).
- `create_ensembles` now accepts a glob string to find datasets. (PR/1081).
- Improved percentile based indicators metadata with the window, threshold and climatology period used to compute percentiles. (GH/1047, PR/1050).

- New `xclim.core.calendar.construct_offset`, the inverse operation of `parse_offset`. (PR/1090).
- Rechunking operations in `xclim.indices.run_length.rle` are now synchronized with dask's options. (PR/1090).
- A convenience recipe for installing key development branches of some dependencies has been added (*$ pip install xclim[upstream]*). (GH/1088, PR/1092).
- A mention of the "missing" checks and options is added to the history attribute of indicators, where appropriate. (GH/1100, PR/1103).

13.1.3 Breaking changes

- `xclim.atmos.water_budget` has been separated into `water_budget` (calculated directly with 'evspsblpot') and `water_budget_from_tas` (original function). (PR/1086).
- Injected parameters in indicators are now left out of a function's signature and will not be included in the history attribute. (PR/1086).
- **The signature for the following Indicators:**
  - cold_spell_duration_index, tg90p, tg10p, tx90p, tx10p, tn90p, tn10p, warm_spell_duration_index, days_over_precip_doy_thresh, days_over_precip_thresh, fraction_over_precip_doy_thresh, fraction_over_precip_thresh, cold_and_dry_days, warm_and_dry_days, warm_and_wet_days, cold_and_wet_days

  have been modified. The parameter for percentiles values is now named after the variable it is supposed to be computed upon. (PR/1050)
- *pytest-runner* has been removed as a dependency (it was never needed for *xclim* development). (PR/1074).
- ***xclim.testing._ utils.py* has been renamed to *xclim.testing.utils.py* for added documentation visibility. (PR/1074).**
  - Some unused functions and classes (`as_tuple`, `TestFile`, `TestDataSet`) have been removed. (PR/1107).

13.1.4 New indicators

- **`universal_thermal_climate_index` and `mean_radiant_temperature` for computing the universal thermal climate index from the near-surface temperature, relative humidity, near-surface windspeed and radiation. (GH/1060, PR/1062).**
  - A new method `ITS90` has also been added for calculating saturation water vapour pressure. (GH/1060, PR/1062).

13.1.5 Internal changes

- Typing syntax has been updated within pre-commit via *isort*. Pre-commit hooks now append *from __future__ import annotations* to all python module imports for backwards compatibility. (GH/1065, PR/1071)
- *isort* project configurations are now set in *setup.cfg*. (PR/1071).
- Many function docstrings, external target links, and internal section references have been adjusted to reduce warnings when building the docs. (PR/1074).

- Code snippets within documentation are now checked and reformatted to *black* conventions with *black-doc*. A *pre-commit* hook is now in place to run these checks. (PR/1098).
- Test coverage statistic no longer includes coverage of the test files themselves. Coverage now reflects lines of usable code covered. (PR/1101).
- Reordered listed authors alphabetically. Promoted @bzah to core contributor. (PR/1105).
- Tests have been added for some functions in *xclim.testing.utils.py*; some previously uncaught bugs in `list_input_variables`, `publish_release_notes`, and `show_versions` have been patched. (GH/1078, PR/1107).

13.1.6 Bug fixes

- Clean the *bias_adjustement* and *history* attributes created by *xclim.sdba.adjust* (e.g. when an argument is an *xr.DataArray*, only print the name instead of the whole array). (GH/1083, PR/1087).
- *pydocstyle* checks were silently failing in the *pre-commit* configuration due to a badly-formed regex. This has been adjusted. (PR/1074).
- *adjust_doy_calendar* was broken when the source or the target were seasonal. (GH/1097, GH/1091, PR/1099)

13.2 v0.36.0 (29-04-2022)

Contributors to this version: Pascal Bourgault (@aulemahal), Juliette Lavoie (@juliettelavoie), David Huard (@huard).

13.2.1 Bug fixes

- Invoking `lazy_indexing` twice in row (or more) using the same indexes (using dask) is now fixed. (GH/1048, PR/1049).
- Filtering out the nans before choosing the first and last values as `fill_value` in `_interp_on_quantiles_1D`. (GH/1056, PR/1057).
- Translations from virtual indicator modules do not override those of the base indicators anymore. (GH/1053, PR/1058).
- Fix mmday unit definition (factor 1000 error). (GH/1061, PR/1063).

13.2.2 New features and enhancements

- `xclim.sdba.measures.rmse` and `xclim.sdba.measures.mae` now use *numpy* instead of *sklearn*. This improves their performances when using *dask*. (PR/1051).
- Argument `append_ends` added to `sdba.unpack_moving_yearly_window` (PR/1059).

13.2.3 Internal changes

- Ipython was unpinned as version 8.2 fixed the previous issue. (GH/1005, PR/1064).

13.3 v0.35.0 (01-04-2022)

Contributors to this version: David Huard (@huard), Trevor James Smith (@Zeitsperre) and Pascal Bourgault (@aulemahal).

13.3.1 New indicators

- New indicator `specific_humidity_from_dewpoint`, computing specific humidity from the dewpoint temperature and air pressure. (GH/864, PR/1027)

13.3.2 New features and enhancements

- New spatial analogues method "szekely_rizzo" (PR/1033).
- Loess smoothing (and detrending) now skip NaN values, instead of propagating them. This can be controlled through the *skipna* argument. (PR/1030).

13.3.3 Bug fixes

- `xclim.analog.spatial_analogs` is now compatible with dask-backed DataArrays. (PR/1033).
- Parameter `dmin` added to spatial analog method "zech_aslan", to avoid singularities on identical points. (PR/1033).
- *xclim* is now compatible with changes in *xarray* that enabled explicit indexing operations. (PR/1038, xarray PR).

13.3.4 Internal changes

- *xclim* now uses the `check-json` and `pretty-format-json` pre-commit checks to validate and format JSON files. (PR/1032).
- The few *logging* artifacts in the `xclim.ensembles` module have been replaced with *warnings.warn* calls or removed. (GH/1039, PR/1044).

13.4 v0.34.0 (25-02-2022)

Contributors to this version: Pascal Bourgault (@aulemahal), Trevor James Smith (@Zeitsperre), David Huard (@huard), Aoun Abel (@bzah).

13.4.1 Announcements

- *xclim* now officially supports Python3.10. (PR/1013).

13.4.2 Breaking changes

- The version pin for *bottleneck* (<1.4) has been lifted. (PR/1013).
- *packaging* has been removed from the *xclim* run dependencies. (PR/1013).
- Quantile mapping adjustment objects (EQM, DQM and QDM) and `sdba.utils.equally_spaced_nodes` will not add additional endpoints to the quantile range. With those endpoints, variables are capped to the reference's range in the historical period, which can be dangerous with high variability in the extremes (ex: pr), especially if the reference doesn't reproduce those extremes credibly. (GH/1015, PR/1016). To retrieve the same functionality as before use:

```
from xclim import sdba

# NQ is the the number of equally spaced nodes, the argument previously given to␣
↪nquantiles directly.
EQM = sdba.EmpiricalQuantileMapping.train(
    ref, hist, nquantiles=sdba.equally_spaced_nodes(NQ, eps=1e-6), ...
)
```

- **The "history" string attribute added by xclim has been modified for readability: (GH/963, PR/1018).**
  - The trailing dot (`.`) was dropped.
  - `None` inputs are now printed as "None" (and not "<NoneType>").
  - Arguments are now always shown as keyword-arguments. This mostly impacts `sdba` functions, as it was already the case for `Indicators`.
- The *cell_methods* string attribute appends only the operation from the indicator itself. In previous version, some indicators also appended the input data's own *cell_method*. The clix-meta importer has been modified to follow the same convention. (GH/983, PR/1022)

13.4.3 New features and enhancements

- *publish_release_notes* now leverages much more regular expression logic for link translations to markdown. (PR/1023).
- Improve performances of percentile bootstrap algorithm by using `xarray.map_block` (GH/932, PR/1017).

13.4.4 Bug fixes

- Loading virtual python modules with `build_indicator_module_from_yaml` is now fixed on some systems where the current directory was not part of python's path. Furthermore, paths of the python and json files can now be passed directly to the `indices` and `translations` arguments, respectively. (GH/1020, PR/1021).

13.4.5 Internal changes

- **Due to an upstream bug in *bottleneck*'s support of virtualenv, *tox* builds for Python3.10 now depend on a patched fork of *bottleneck*. This workaround will be removed once the fix is merged upstream. (PR/1013, see: bottleneck PR/397).**
  - This has been removed with the release of bottleneck version 1.3.4. (PR/1025).
- GitHub CI actions now use the deadsnakes python PPA Action for gathering the Python3.10 development headers. (PR/1013).
- The "is_dayofyear" attribute added by several indices is now a `numpy.int32` instance, instead of python's `int`. This ensures a THREDDS server can read it when the variable is saved to a netCDF file with *xarray/netCDF4-python*. (GH/980, PR/1019).
- The *xclim* git repository now offers Issue Forms for some general issue types.

13.5 v0.33.2 (2022-02-09)

Contributors to this version: Pascal Bourgault (@aulemahal), Juliette Lavoie (@juliettelavoie), Trevor James Smith (@Zeitsperre).

13.5.1 Announcements

- *xclim* no longer supports Python3.7. Code conventions and new features for Python3.8 (PEP 569) are now accepted. (GH/966, PR/1000).

13.5.2 Breaking changes

- Python3.7 (PEP 537) support has been officially deprecated. Continuous integration testing is no longer run against this version of Python. (GH/966, PR/1000).

13.5.3 Bug fixes

- Adjusted behaviour in `dataflags.ecad_compliant` to remove *data_ vars* of invalids checks that return *None*, causing issues with *dask*. (PR/1002).
- Temporarily pinned *ipython* below version 8.0 due to behaviour causing hangs in GitHub Actions and ReadTheDocs. (GH/1005, PR/1006).
- `indices.stats` methods where adapted to handle dask-backed arrays. (GH/1007, :*pull:`1011*).
- `sdba.utils.interp_on_quantiles`, with `extrapolation='constant'`, now interpolates the limits of the interpolation along the time grouping index, fixing a issue with "time.month" grouping. (GH/1008, PR/1009).

13.5.4 Internal changes

- *pre-commit* now uses Black 22.1.0 with Python3.8 style conventions. Existing code has been adjusted. (PR/1000).
- *tox* builds for Python3.7 have been deprecated. (PR/1000).
- Docstrings and documentation has been adjusted for grammar and typos. (PR/1000).
- `sdba.utils.extrapolate_qm` has been removed, as announced for xclim 0.33. (PR/1009).

13.6 v0.33.0 (2022-01-28)

Contributors to this version: Trevor James Smith (@Zeitsperre), Pascal Bourgault (@aulemahal), Tom Keel (@Thomasjkeel), Jeremy Fyke (@JeremyFyke), David Huard (@huard), Abel Aoun (@bzah), Juliette Lavoie (@juliettelavoie), Yannick Rousseau (@yrouranos).

13.6.1 Announcements

- Deprecation: Release 0.33.0 of *xclim* will be the last version to explicitly support Python3.7 and *xarray<0.21.0*.
- *xclim* now requires yaml files to pass *yamllint* checks on Pull Requests. (PR/981).
- ***xclim* now requires docstrings have valid ReStructuredText formatting to pass basic linting checks. (PR/993). Checks generally require:**
  - Working hyperlinks and reference tags.
  - Valid content references (e.g. *:py:func:*).
  - Valid NumPy-formatted docstrings.
- The *xclim* developer community has now adopted the 'Contributor Covenant' Code of Conduct v2.1 (text). (GH/948, PR/996).

13.6.2 New indicators

- `jetstream_metric_woollings` indicator returns latitude and strength of jet-stream in u-wind field. (GH/923, PR/924).

13.6.3 New features and enhancements

- **Features added and modified to allow proper multivariate adjustments. (PR/964).**
  - Added `xclim.sdba.processing.to_additive_space` and `xclim.sdba.processing.from_additive_space` to transform "multiplicative" variables to the additive space. An example of multivariate adjustment using this technique was added to the "Advanced" sdba notebook.
  - `xclim.sdba.processing.normalize` now also returns the norm. `xclim.sdba.processing.jitter` was created by combining the "under" and "over" methods.
  - `xclim.sdba.adjustment.PrincipalComponent` was modified to have a simpler signature. The "full" method for finding the best PC orientation was added. (GH/697).

- New `xclim.indices.stats.parametric_cdf` function to facilitate the computation of return periods over DataArrays of statistical distribution parameters (GH/876, PR/984).
- Add `copy` parameter to `percentile_doy` to control if the array input can be dumped after computing percentiles (GH/932, PR/985).
- New improved algorithm for `dry_spell_total_length`, performing the temporal indexing at the right moment and with control on the aggregation operator (`op`) for determining the dry spells.
- Added `properties.py` and `measures.py` in order to perform diagnostic tests of sdba (GH/424, PR/967).
- Update how `percentile_doy` rechunk the input data to preserve the initial chunk size. This should make the computation memory footprint more predictable (GH/932, PR/987).

13.6.4 Breaking changes

- To reduce import complexity, *select_time* has been refactored/moved from `xclim.indices.generic` to `xclim.core.calendar`. (GH/949, PR/969).
- The stacking dimension of `xclim.sdba.stack_variables` has been renamed to "multivar" to avoid name conflicts with the "variables" property of xarray Datasets. (PR/964).
- *xclim* now requires *cf-xarray>=0.6.1*. (GH/923, PR/924).
- *xclim* now requires *statsmodels*. (GH/424, PR/967).

13.6.5 Internal changes

- Added a CI hook in `.pre-commit-config.yaml` to perform automated *pre-commit* corrections with GitHub CI. (PR/965).
- Adjusted CI hooks to fail earlier if *lint* checks fail. (PR/972).
- *TrainAdjust* and *Adjust* object have a new *skip_input_checks* keyword arg to their *train* and *adjust* methods. When *True*, all unit-, calendar- and coordinate-related input checks are skipped. This is an ugly solution to disappearing attributes when using *xr.map_blocks* with dask. (PR/964).
- **Some slow tests were marked *slow* to help speed up the standard test ensemble. (PR/969).**
  - Tox testing ensemble now also reports slowest tests using the `--durations` flag.
- *pint* no longer emits warnings about redefined units when the *logging* module is loaded. (GH/990, PR/991).
- Added a CI step for cancelling running workflows in pull requests that receive multiple pushes. (PR/988).

13.6.6 Bug fixes

- Fix mistake in the units of spell_length_distribution. (GH/1003, PR/1004)

13.7 v0.32.1 (2021-12-17)

13.7.1 Bug fixes

- Adjusted a test (`test_cli::test_release_notes`) that prevented conda-forge test ensemble from passing. (PR/962).

13.8 v0.32.0 (2021-12-17)

Contributors to this version: Pascal Bourgault (@aulemahal), Travis Logan (@tlogan2000), Trevor James Smith (@Zeitsperre), Abel Aoun (@bzah), David Huard (@huard), Clair Barnes (@clairbarnes), Raquel Alegre (@raquel-ucl), Jamie Quinn (@JamieJQuinn), Maliko Tanguy (@malngu), Aaron Spring (@aaronspring).

13.8.1 Announcements

- **Code coverage (*coverage/coveralls*) is now a required CI check for merging Pull Requests. Requirements are now:**
  - No individual run may report *<80%* code coverage.
  - Some drop in coverage is now tolerable, but runs cannot dip below *-0.25%* relative to the main branch.

13.8.2 New features and enhancements

- Added an optimized pathway for `xclim.indices.run_length` functions when `window=1`. (PR/911, GH/910).
- The data input frequency expected by `Indicator` is now in the `src_freq` attribute and is thus controllable by subclassing existing indicators. (GH/898, PR/927).
- New `**indexer` keyword args added to many indicators, it accepts the same arguments as `xclim.indices.generic.select_time`, which has been improved. Unless otherwise specified, the time selection is done before any computation. (PR/934, GH/899).
- Rewrite of `xclim.sdba.ExtremeValues`, now fixed with a correct algorithm. It has not been tested extensively and should be considered experimental. (PR/914, GH/789, GH/790).
- Added *days_over_precip_doy_thresh* and *fraction_over_precip_doy_thresh* indicators to distinguish between WMO and ECAD definition of the Rxxp and RxxpTot indices. (GH/931, PR/940).
- Update *xclim.core.utils.nan_calc_percentiles* to improve maintainability. (PR/942).
- Added *heat_index* indicator. Added *heat_index* indicator. This is similar to *humidex* but uses a different dew point as well as heat balance equations which account for variables other than vapor pressure. (GH/807) and (PR/915).
- Added alternative method for `xclim.indices.potential_evapotranspiration` based on *mcguinness-bordne05* (from Tanguay et al. 2018). (PR/926, GH/925).

- Added *snw_max* and *snw_max_doy* indicators to compute the maximum snow amount and the day of year of the maximum snow amount respectively. (GH/776, PR/950).
- Added index for calculating ratio of convective to total precipitation. (GH/920, PR/921).
- Added *wetdays_prop* indicator to calculate the proportion of days in a period where the precipitation is greater than a threshold. (PR/919, GH/918).

13.8.3 Breaking changes

- Following version 1.9 of the CF Conventions, published in September 2021, the calendar name "gregorian" is deprecated. `core.calendar.get_calendar` will return "standard", even if the underlying cftime objects still use "gregorian" (cftime <= 1.5.1). (PR/935).
- `xclim.sdba.utils.extrapolate_qm` is now deprecated and will be removed in version 0.33. (PR/941).
- Dependency `pint` minimum necessary version is now 0.10. (PR/959).

13.8.4 Internal changes

- Removed some logging configurations in `xclim.core.dataflags` that were polluting python's main logging configuration. (PR/909).
- Synchronized logging formatters in `xclim.ensembles` and `xclim.core.utils`. (PR/909).
- Added a helper function for generating the release notes with dynamically-generated ReStructuredText or Markdown-formatted hyperlinks (PR/922, GH/907).
- Split of resampling-related functionality of `Indicator` into new `ResamplingIndicator` and `ResamplingIndicatorWithIndexing` subclasses. The use of new (private) methods makes it easier to inject functionality in indicator subclasses. (GH/867, PR/927, PR/934).
- French translation metadata fields are now cleaner and much more internally consistent, and many empty metadata fields (e.g. `comment_fr`) have been removed. (PR/930, GH/929).
- Adjustments to the `tox` builds so that slow tests are now run alongside standard tests (for more accurate coverage reporting). (PR/938).
- Use `xarray.apply_ufunc` to vectorize statistical functions. (PR/943).
- Refactor of `xclim.sdba.utils.interp_on_quantiles` so that it now handles the extrapolation directly and to better handle missing values. (PR/941).
- Updated *heating_degree_days* and *fraction_over_precip_thresh* documentations. (GH/952, PR/953).
- Added an intersphinx mapping to xarray. (PR/955).
- Added a CodeQL security analysis GitHub CI hook on push to master and on Friday nights. (PR/960).

13.8.5 Bug fixes

- Fix bugs in the *cf_attrs* and/or *abstract* of *continuous_snow_cover_end* and *continuous_snow_cover_start*. (PR/908).
- Remove unnecessary *keep_attrs* from *resample* call which would raise an error in futur Xarray version. (PR/937).
- Fixed a bug in the regex that parses usernames in the history. (PR/945).
- Fixed a bug in `xclim.indices.generic.doymax` and `xclim.indices.generic.doymin` that prevented the use of the functions on multidimensional data. (PR/950, GH/951).
- Skip all missing values in `xclim.sdba.utils.interp_on_quantiles`, drop them from both the old and new coordinates, as well as from the old values. (PR/941).
- "degrees_north" and "degrees_east" (and their variants) are now considered independent units, so that `pint` and `xclim.core.units.ensure_cf_units` don't convert them to "deg". (PR/959).
- Fixed a bug in `xclim.core.dataflags` that would misidentify the "extra" variable to be called when running multivariate checks. (PR/957, GH/861).

13.9 v0.31.0 (2021-11-05)

Contributors to this version: Abel Aoun (@bzah), Pascal Bourgault (@aulemahal), David Huard (@huard), Juliette Lavoie (@juliettelavoie), Travis Logan (@tlogan2000), Trevor James Smith (@Zeitsperre).

13.9.1 New indicators

- `thawing_degree_days` indicator returns degree-days above a default of *thresh="0 degC"*. (PR/895, GH/887).
- `freezing_degree_days` indicator returns degree-days below a default of *thresh="0 degC"*. (PR/895, GH/887).
- **Several frost-free season calculations are now available as both indices and indicators. (PR/895, GH/887):**
  - `frost_free_season_start`
  - `frost_free_season_end`
  - `frost_free_season_length`
- `growing_season_start` is now offered as an indice and as an indicator to complement other growing season-based indicators (threshold calculation with *op=">="*). (PR/895, GH/887).

13.9.2 New features and enhancements

- Improve cell_methods checking to search the wanted method within the whole string. (PR/866, GH/863).
- New `align_on='random` option for `xclim.core.calendar.convert_calendar`, for conversions involving '360_day' calendars. (PR/875, GH/841).
- `dry_spell_frequency` now has a parameter *op: {"sum", "max"}* to choose if the threshold is compared against the accumulated or maximal precipitation, over the given window. (PR/879).
- `maximum_consecutive_frost_free_days` is now checking that the minimum temperature is above or equal to the threshold (instead of only above). (PR/883, GH/881).
- The ANUCLIM virtual module has been updated to accept weekly and monthly inputs and with improved metadata. (PR/885, GH/538)
- The `sdba.loess` algorithm has been optimized to run faster in all cases, with an even faster special case (`equal_spacing=True`) when the x coordinate is equally spaced. When activated, this special case might return results different from without, up to around 0.1%. (PR/865).
- Add support for group's window and additional dimensions in `LoessDetrend`. Add new `RollingMeanDetrend` object. (PR/865).
- Missing value algorithms now try to infer the source timestep of the input data when it is not given. (PR/885).
- On indices, *bootstrap* parameter documentation has been updated to explain when and why it should be used. (PR/893, GH/846).

13.9.3 Breaking changes

- Major changes in the YAML schema for virtual submodules, now closer to how indicators are declared dynamically, see the doc for details. (PR/849, GH/848).
- Removed `xclim.generic.daily_downsampler`, as it served no purpose now that xarray's resampling works with cftime (PR/888, GH/889).
- Refactor of `xclim.core.calendar.parse_offset`, output types were changed to useful ones (PR/885).
- **Major changes on how parameters are passed to indicators. (PR/873):**
  - Their signature is now consistent : input variables (DataArrays, optional or not) are positional or keyword arguments and all other parameters are keyword only. (GH/855, GH/857)
  - Some indicators have modified signatures because we now rename variables when wrapping generic indices. This is the case for the whole cf module, for example.
  - `Indicator.parameters` is now a property generated from `Indicator._all_parameters`, as the latter includes the injected parameters. The keys of the former are instances of new `xclim.core.indicator.Parameter`, and not dictionaries as before.
  - New `Indicator.injected_parameters` to see which compute function arguments will be injected at call time.
  - See the pull request (PR/873) for all information.
- The call signature for `huglin_index` has been modified to reflect the correct variables used in its formula (*tasmin* -> *tas*; *thresh_tasmin* -> *thresh*). (PR/903, GH/902).

13.9.4 Internal changes

- Pull Request contributions now require hyperlinks to the issue and pull request pages on GitHub listed alongside changess in HISTORY.rst. (PR/860, GH/854).
- Updated the contribution guidelines to better give credit to contributors and more easily track changes. (PR/869, GH/868).
- Enabled coveralls code coverage reporting for GitHub CI. (PR/870).
- Added automated TestPyPI and PyPI-publishing workflows for GitHub CI. (PR/872).
- Changes on how indicators are constructed. (PR/873).
- Added missing algorithms tests for conversion from hourly to daily. (PR/888).
- Updated pre-commit hooks to use black v21.10.b0. (PR/896).
- Moved `stack_variables`, `unstack_variables`, `construct_moving_yearly_window` and `unpack_moving_yearly_window` from `xclim.sdba.base` to `xclim.sdba.processing`. They still are imported in `xclim.sdba` as before. (PR/892).
- Many improvements to the documentation. (PR/892, GH/880).
- Added regex replacement handling in setup.py to facilitate publishing contributor/contribution links on PyPI. (PR/906).

13.9.5 Bug fixes

- Fix a bug in bootstrapping where computation would fail when the dataset time coordinate is encoded using *cftime.datetime*. (PR/859).
- Fix a bug in `build_indicator_module_from_yaml` where bases classes (Daily, Hourly, etc) were not usable with the *base* field. (PR/885).
- `percentile_doy` alpha and beta parameters are now properly transmitted to bootstrap calls of this function. (PR/893, GH/846).
- When called with a 1D da and ND index, `xclim.indices.run_length.lazy_indexing` now drops the auxiliary coordinate corresponding to da's index. This fixes a bug with ND data in `xclim.indices.run_length.season`. (PR/900).
- Fix name of heating degree days in French (*"chauffe"* -> "*chauffage*"). (PR/895).
- Corrected several French indicator translation description strings (bad usages of *"."* in *description* and *long_ name* fields). (PR/895).
- Fixed an error with the formula for `huglin_index` where *tasmin* was being used in the calculation instead of *tas*. (PR/903, GH/902).

13.10 v0.30.1 (2021-10-01)

13.10.1 Bug fixes

- Fix a bug in `xclim.sdba`'s `map_groups` where 1D input including an auxiliary coordinate would fail with an obscure error on a reducing operation.

13.11 v0.30.0 (2021-09-28)

13.11.1 New indicators

- `climatological_mean_doy` indice returns the mean and standard deviation across a climatology according to day-of-year (*xarray.DataArray.groupby("time.dayofyear")*). A moving window averaging of days can also be supplied (default:*window=1*).
- `within_bnds_doy` indice returns a boolean array indicating whether or not array's values are within bounds for each day of the year.
- Added `atmos.wet_precip_accumulation`, an indicator accumulating precipitation over wet days.
- Module ICCLIM now includes `PRCPTOT`, which accumulates precipitation for days with precipitation above 1 mm/day.

13.11.2 New features and enhancements

- `xclim.core.utils.nan_calc_percentiles` now uses a custom algorithm instead of `numpy.nanpercentiles` to have more flexibility on the interpolation method. The performance is also improved.
- `xclim.core.calendar.percentile_doy` now uses the 8th method of Hyndman & Fan for linear interpolation (alpha = beta = 1/3). Previously, the function used Numpy's percentile, which corresponds to the 7th method. This change is motivated by the fact that the 8th is recommended by Hyndman & Fay and it ensures consistency with other climate indices packages (*climdex*, *icclim*). Using *alpha = beta = 1* restores the previous behaviour.
- `xclim.core.utils._cal_perc` is now only a proxy for `xc.core.utils.nan_calc_percentiles` with some axis moves.
- ***xclim* now implements many data quality assurance flags (`xclim.core.dataflags`) for temperature and precipitation based on ICCLIM documentation guidelines. These checks include the following:**
  - Temperature (variables: `tas`, `tasmin`, `tasmax`): `tasmax_below_tasmin`, `tas_exceeds_tasmax`, `tas_below_tasmin`, `temperature_extremely_low` (*thresh="-90 degC"*), `temperature_extremely_high` (*thresh="60 degC"*).
  - Precipitation-specific (variables: `pr`, `prsn`,): `negative_accumulation_values`, `very_large_precipitation_events` (*thresh="300 mm d-1"*).
  - Wind-specific (variables: `sfcWind`, `wsgsmax`/`sfcWindMax`): `wind_values_outside_of_bounds`
  - Generic: `outside_n_standard_deviations_of_climatology`, `values_repeating_for_n_or_more_days`, `values_op_thresh_repeating_for_n_or_more_days`, `percentage_values_outside_of_bounds`.

These quality-assurance checks are selected according to CF-standard variable names, and can be triggered via `xclim.core.dataflags.data_flags(xarray.DataArray, xarray.Dataset)`. These checks are separate from the Indicator-defined *datachecks* and must be launched manually. They'll return an array of data_flags as boolean variables. If called with *raise_flags=True*, will raise an Exception with comments for each quality control check raised.

- A convenience function (`xclim.core.dataflags.ecad_compliant`) is also offered as a method for asserting that data adheres to all relevant ECAD/ICCLIM checks. For more information on usage, consult the docstring/documentation.
- A new utility "`dataflags`" is also available for performing fast quality control checks from the command-line (*xclim dataflags –help*). See the CLI documentation page for usage examples.
- Added missing typed call signatures, expected returns and docstrings for many `xclim.core.calendar` functions.

13.11.3 Breaking changes

- All "ANUCLIM" indices and indicators have lost their *src_timestep* argument. Most of them were not using it and now every function infers the frequency from the data directly. This may add stricter constraints on the time coordinate, the same as for `xarray.infer_freq`.
- Many functions found within `xclim.core.cfchecks` (`generate_cfcheck` and `check_valid_*`) have been removed as existing indicator CF-standard checks and data checks rendered them redundant/obsolete.

13.11.4 Bug fixes

- Fixes in `sdba` for (1) inputs with dimensions without coordinates, for (2) `sdba.detrending.MeanDetrend` and for (3) `DetrendedQuantileMapping` when used with dask's distributed scheduler.
- Replaced instances of '◦' ("White bullet") with '°' ("Degree Sign") in `icclim.yaml` as it was causing issues for non-UTF8 environments.
- Addressed an edge case where `test_sdba::test_standardize` randomness could generate values that surpass the test error tolerance.
- Added a missing *.txt* file to the MANIFEST of the source distributable in order to be able to run all tests.
- `xc.core.units.rate2amount` is now exact when the sampling frequency is monthly, seasonal or yearly. Earlier, monthly and yearly data were computed using constant month and year length. End-of-period frequencies are also correctly understood (ex: "M" vs "MS").
- In the `potential_evapotranspiration` indice, add abbreviated `method` names to docstring.
- Fixed an issue that prevented using the default `group` arg in adjustment objects.
- Fix bug in `missing_wmo`, where a period would be considered valid if all months met WMO criteria, but complete months in a year were missing. Now if any month does not meet criteria or is absent, the period will be considered missing.
- Fix bootstrapping with dask arrays. Dask does not support using `loc` with multiple indexes to set new values so a workaround was necessary.
- Fix bootstrapping when the bootstrapped year must be converted to a 366_day calendar.
- Virtual modules and translations now use 'UTF-8' by default when reading yaml or json file, instead of a machine-dependent encoding.

13.11.5 Internal Changes

- *xclim* code quality checks now use the newest *black* (v21.8-beta). Checks launched via *tox* and *pre-commit* now run formatting modifications over Jupyter notebooks found under *docs*.

13.12 v0.29.0 (2021-08-30)

13.12.1 Announcements

- It was found that the `ExtremeValues` adjustment algorithm was not as accurate and stable as first thought. It is now hidden from `xclim.sdba` but can still be accessed via `xclim.sdba.adjustment`, with a warning. Work on improving the algorithm is ongoing, and a better implementation will be in a future version.
- It was found that the `add_dims` argument of `sdba.Grouper` had some caveats throughout `sdba`. This argument is to be used with care before a careful analysis and more testing is done within `xclim`.

13.12.2 Breaking changes

- *xclim* has switched back to updating the `history` attribute (instead of `xclim_history`). This impacts all indicators, most ensemble functions, `percentile_doy` and `sdba.processing` (see below).
- Refactor of `sdba.processing`. Now all functions take one or more DataArrays as input, plus some parameters. And output one or more dataarrays (not Datasets). Units and metadata is handled. This impacts `sdba.processing.adapt_freq` especially.
- Add unit handling in `sdba`. Most parameters involving quantities are now expecting strings (and not numbers). Adjustment objects will ensure ref, hist and sim all have the same units (taking ref as reference).
- The Adjustment` classes of `xclim.sdba` have been refactored into 2 categories:
  - `TrainAdjust` objects (most of the algorithms), which are created **and** trained in the same call: `obj = Adj.train(ref, hist, **kwargs)`. The `.adjust` step stays the same.
  - `Adjust` objects (only `NpdfTransform`), which are never initialized. Their `adjust` class method performs all the work in one call.
- `snowfall_approximation` used a "<" condition instead of "<=" to determine the snow fraction based on the freezing point temperature. The new version sticks to the convention used in the Canadian Land Surface Scheme (CLASS).
- Removed the *"gis"*, *"docs"*, *"test"* and *"setup"`extra dependencies from ``setup.py*`. The `dev` recipe now includes all tools needed for xclim's development.

13.12.3 New features and enhancements

- `snowfall_approximation` has gained support for new estimation methods used in CLASS: 'brown' and 'auer'.
- A `ValidationError` will be raised if temperature units are given as 'deg C', which is misinterpreted by pint.
- Functions computing run lengths (sequences of consecutive *"True"* values) now take the `index` argument. Possible values are `first` and `last`, indicating which item in the run should be used to index the run length. The default is set to *"first"*, preserving the current behavior.
- New `sdba_encode_cf` option to workaround a cftime/xarray performance issue when using dask.

13.12.4 New indicators

- `effective_growing_degree_days` indice returns growing degree days using dynamic start and end dates for the growing season (based on Bootsma et al. (2005)). This has also been wrapped as an indicator.
- `qian_weighted_mean_average` (based on Qian et al. (2010)) is offered as an alternate method for determining the start date using a weighted 5-day average (`method="qian"`). Can also be used directly as an indice.
- `cold_and_dry_days` indicator returns the number of days where the mean daily temperature is below the 25th percentile and the mean daily precipitation is below the 25th percentile over period. Added as `CD` indicator to ICCLIM module.
- `warm_and_dry_days` indicator returns the number of days where the mean daily temperature is above the 75th percentile and the mean daily precipitation is below the 25th percentile over period. Added as `WD` indicator to ICCLIM module.
- `warm_and_wet_days` indicator returns the number of days where the mean daily temperature is above the 75th percentile and the mean daily precipitation is above the 75th percentile over period. Added as `WW` indicator to ICCLIM module.
- `cold_and_wet_days` indicator returns the number of days where the mean daily temperature is below the 25th percentile and the mean daily precipitation is above the 75th percentile over period. Added as `CW` indicator to ICCLIM module.
- `calm_days` indicator returns the number of days where surface wind speed is below threshold.
- `windy_days` indicator returns the number of days where surface wind speed is above threshold.

13.12.5 Bug fixes

- **Various bug fixes in bootstrapping:**
  - in `percentile_bootstrap` decorator, fix the popping of bootstrap argument to propagate in to the function call.
  - in `bootstrap_func`, fix some issues with the resampling frequency which was not working when anchored.
- Made argument `thresh` of `sdba.LOCI` required, as not giving it raised an error. Made defaults explicit in the adjustments docstrings.
- Fixes in `sdba.processing.adapt_freq` and `sdba.nbutils.vecquantiles` when handling all-nan slices.

- Dimensions in a grouper's `add_dims` are now taken into consideration in function wrapped with `map_blocks/groups`. This feature is still not fully tested throughout `sdba` though, so use with caution.
- Better dtype preservation throughout `sdba`.
- "constant" extrapolation in the quantile mappings' adjustment is now padding values just above and under the target's max and min, instead of ±`np.inf`.
- Fixes in `sdba.LOCI` for the case where a grouping with additionnal dimensions is used.

13.12.6 Internal Changes

- The behaviour of `xclim.testing._utils.getfile` was adjusted to launch file download requests for web-hosted md5 files for every call to compare against local test data. This was done to validate that locally-stored test data is identical to test data available online, without resorting to git-based actions. This approach may eventually be revised/optimized in the future.

13.13 v0.28.1 (2021-07-29)

13.13.1 Announcements

- The *xclim* binary package available on conda-forge will no longer supply `clisops` by default. Installation of `clisops` must be performed explicitly to preserve subsetting and bias correction capabilities.

13.13.2 New indicators

- `snow_depth` indicator returns the mean snow depth over period. Added as `SD` to ICCLIM module.

13.13.3 Internal Changes

- Minor modifications to many function call signatures (type hinting) and docstrings (numpy docstring compliance).

13.14 v0.28.0 (2021-07-07)

13.14.1 New features and enhancements

- Automatic load of translations on import and possibility to pass translations for virtual modules.
- New `xclim.testing.list_datasets` function listing all available test datasets in repo `xclim-testdata`.
- `spatial_analogs` accepts multi-indexes as the `dist_dim` parameter and will work with candidates and target arrays of different lengths.
- `humidex` can be computed using relative humidity instead of dewpoint temperature.
- New `sdba.construct_moving_yearly_window` and `sdba.unpack_moving_yearly_window` for moving window adjustments.

- New `sdba.adjustment.NpdfTransform` which is an adaptation of Alex Cannon's version of Pitié's *N-dimensional probability density function transform*. Uses new `sdba.utils.rand_rot_matrix`. *Experimental, subject to changes.*
- New `sdba.processing.standardize`, `.unstandardize` and `.reordering`. All of them, tools needed to replicate Cannon's MBCn algorithm.
- New `sdba.processing.escore`, backed by `sdba.nbutils._escore` to evaluate the performance of the N pdf transform.
- New function `xclim.indices.clausius_clapeyron_scaled_precipitation` can be used to scale precipitation according to changes in mean temperature.
- Percentile based indices gained a `bootstrap` argument that applies a bootstrapping algorithm to reduce biases on exceedance frequencies computed over *in base* and *out of base* periods. *Experimental, subject to changes.*
- Added a *.zenodo.json* file for collecting and maintaining author order and tracking ORCIDs.

13.14.2 Bug fixes

- Various bug fixes in sdba :
  - in `QDM.adjust`, fix bug occurring with coords of 'object' dtype and `interp='nearest'`.
  - in `nbutils.quantiles`, fix dtype bug when using `float32` data.
  - raise a proper error when `ref` and `hist` have a different calendar for map_blocks-backed adjustments.

13.14.3 Breaking changes

- `spatial_analogs` does not support sequence of `dist_dim` anymore. Users are responsible for stacking dimensions prior to calling `spatial_analogs`.

13.14.4 New indicators

- `biologically_effective_degree_days` (with `method="gladstones"`) indice computes degree-days between two specific dates, with a capped daily max value as well as latitude and temperature range swing as modifying coefficients (based on Gladstones, J. (1992)). This has also been wrapped as an indicator.
- An alternative implementation of `biologically_effective_degree_days` (with `method="icclim"`, based on ICCLIM formula) ignores latitude and temperature range swing modifiers and uses an alternate `end_date`. Wrapped and available as an ICCLIM indicator.
- `cool_night_index` indice returns the mean minimum temperature in September (`lat >= 0` deg N) or March (`lat < 0` deg N), based on Tonietto & Carbonneau, 2004 (10.1016/j.agrformet.2003.06.001). Also available as an indicator (see indices *Notes* section on indicator usage recommendations).
- `latitude_temperature_index` indice computes LTI values based on mean temperature of warmest month and a parameterizable latitude coefficient (default: `lat_factor=75`) based on Jackson & Cherry, 1988, and Kenny & Shao, 1992 (10.1080/00221589.1992.11516243). This has also been wrapped as an indicator.

- `huglin_index` indice computes Huglin Heliothermal Index (HI) values based on growing degrees and a latitude-influenced coefficient for day-length (based on Huglin. (1978)). The indice supports several methods of estimating the latitude coefficient:
  - `method="smoothed"`: Marks latitudes between -40 N and 40 N with `k=1`, and linearly increases to `k=1.06` at `|lat|==50`.
  - `method="icclim"`: Uses a stepwise function based on the the original method as presented by Huglin (1978). Identical to the ICCLIM implementation.
  - `method="jones"`: Uses a more robust calculation for calculating day-lengths, based on Hall & Jones (2010). This method is now also available for `biologically_effective_degree_days`.
- The generic indice `day_length`, used for calculating approximate daily day-length in hours per day or, given `start_date` and `end_date`, the total aggregated day-hours over period. Uses axial tilt, start and end dates, calendar, and approximate date of northern hemisphere summer solstice, based on Hall & Jones (2010).

13.14.5 Internal Changes

- `aggregate_between_dates` (introduced in v0.27.0) now accepts `DayOfYear`-like strings for supplying start and end dates (e.g. `start="02-01", end="10-31"`).
- The indicator call sequence now considers "variable" the inputs annoted so. Dropped the `nvar` attribute.
- Default cfcheck is now to check metadata according to the variable name, using CMIP6 names in xclim/data/variable.yml.
- `Indicator.missing` defaults to "skip" if `freq` is absent from the list of parameters.
- Minor modifications to the GitHub Pull Requests template.
- Simplification of some yaml elements for virtual modules.
- Allow injecting `freq` without the missing checks failing.

13.15 v0.27.0 (2021-05-28)

13.15.1 New features and enhancements

- Rewrite of nearly all adjustment methods in `sdba`, with use of `xr.map_blocks` to improve scalability with dask. Rewrite of some parts of the algorithms with numba-accelerated code.
- "GFWED" specifics for fire weather computation implemented back into the FWI module. Outputs are within 3% of GFWED data.
- Addition of the *run_length_ufunc* option to control which run length algorithm gets run. Defaults stay the same (automatic switch dependent of the input array : the 1D version is used with non-dask arrays with less than 9000 points per slice).
- Indicator modules built from YAML can now use custom indices. A mapping or module of them can be given to `build_indicator_module_from_yaml` with the `indices` keyword.
- Virtual submodules now include an *iter_indicators* function to iterate over the pairs of names and indicator objects in that module.
- The indicator string formatter now accepts a "r" modifier which passes the raw strings instead of the adjective version.

- Addition of the *sdba_extra_output* option to adds extra diagnostic variables to the outputs of Adjustment objects. Implementation of *sim_q* in QuantileDeltaMapping and *nclusters* in ExtremeValues.

13.15.2 Breaking changes

- The *tropical_nights* indice is being deprecated in favour of *tn_days_above* with `thresh="20 degC"`. The indicator remains valid, now wrapping this new indice.
- Results of `sdba.Grouper.apply` for `Grouper``s without a group (ex: ``Grouper('time')`) will contain a `group` singleton dimension.
- The *daily_freezethaw_cycles* indice is being deprecated in favour of `multiday_temperature_swing` with temp thresholds at 0 degC and `window=1, op="sum"`. The indicator remains valid, now wrapping this new indice.
- CMIP6 variable names have been adopted whenever possible in xclim. Changes are:
  - `swe` is now `snw` (`snw` is the snow amount [kg / m^2] and `swe` the liquid water equivalent thickness [m])
  - `rh` is now `hurs`
  - `dtas` is now `tdps`
  - `ws` (in FWI) is now `sfcWind`
  - `sic` is now `siconc`
  - `area` (of sea ice indicators) is now `areacello`
  - Indicators `RH` and `RH_FROMDEWPOINT` have be renamed to `HURS` and `HURS_FROMDEWPOINT`. These are changes in the _identifiers_, the python names (`relative_humidity[...]`) are unchanged.

13.15.3 New indicators

- *atmos.corn_heat_units* computes the daily temperature-based index for corn growth.
- New indices and indicators for *tx_days_below*, *tg_days_above*, *tg_days_below*, and *tn_days_above*.
- *atmos.humidex* returns the Canadian *humidex*, an indicator of perceived temperature account for relative humidity.
- *multiday_temperature_swing* indice for returning general statistics based on spells of doubly-thresholded temperatures (Tmin < T1, Tmax > T2).
- New indicators *atmos.freezethaw_frequency*, *atmos.freezethaw_spell_mean_length*, *atmos.freezethaw_spell_max_length* for statistics of Tmin < 0 degC and Tmax > 0 deg C days now available (wrapped from *multiday_temperature_swing*).
- *atmos.wind_chill_index* computes the daily wind chill index. The default is similar to what Environment and Climate Change Canada does, options are tunable to get the version of the National Weather Service.

13.15.4 Internal Changes

- *run_length.rle_statistics* now accepts a *window* argument.
- Common arguments to the *op* parameter now have better adjective and noun formattings.
- Added and adjusted typing in call signatures and docstrings, with grammar fixes, for many *xclim.indices* operations.
- Added internal function `aggregate_between_dates` for array aggregation operations using xarray datetime arrays with start and end DayOfYear values.

13.16 v0.26.1 (2021-05-04)

- Bug fix release adding *ExtremeValues* to publicly exposed bias-adjustment methods.

13.17 v0.26.0 (2021-04-30)

13.17.1 Announcements

- *xclim* no longer supports Python3.6. Code conventions and new features from Python3.7 (PEP 537 Features) are now accepted.

13.17.2 New features and enhancements

- *core.calendar.doy_to_days_since* and *days_since_to_doy* to allow meaningful statistics on doy data.
- New bias second-order adjustment method "ExtremeValues", intended for re-adjusting extreme precipitation values.
- Virtual indicators modules can now be built from YAML files.
- Indicators can now be built from dictionaries.
- New generic indices, implementation of *clix-meta*'s index functions.
- On-the-fly generation of climate and forecasting convention (CF) checks with *xc.core.cfchecks.generate_cfcheck*, for a few known variables only.
- New *xc.indices.run_length.rle_statistics* for min, max, mean, std (etc) statistics on run lengths.
- New virtual submodule *cf*, with CF standard indices defined in clix-meta.
- Indices returning day-of-year data add two new attributes to the output: *is_dayofyear* (=1) and *calendar*.

13.17.3 Breaking changes

- *xclim* now requires *xarray>=0.17*.
- Virtual submodules *icclim* and *anuclim* are not available at the top level anymore (only through *xclim.indicators*).
- Virtual submodules *icclim* and *anuclim* now provide *Indicators* and not indices.
- Spatial analog methods "KLDIV" and "Nearest Neighbor" now require *scipy>=1.6.0*.

13.17.4 Bug fixes

- *from_string* object creation in sdba has been removed. Now replaced with use of a new dependency, *jsonpickle*.

13.17.5 Internal Changes

- *pre-commit* linting checks now run formatting hook *black==21.4b2*.
- Code cleaning (more accurate call signatures, more use of https links, docstring updates, and typo fixes).

13.18 v0.25.0 (2021-03-31)

13.18.1 Announcements

- Deprecation: Release 0.25.0 of *xclim* will be the last version to explicitly support Python3.6 and *xarray<0.17.0*.

13.18.2 New indicators

- *land.winter_storm* computes days with snow accumulation over threshold.
- *land.blowing_snow* computes days with both snow accumulation over last days and high wind speeds.
- *land.snow_melt_we_max* computes the maximum snow melt over n days, and *land.melt_and_precip_max* the maximum combined snow melt and precipitation.
- *snd_max_doy* returns the day of the year where snow depth reaches its maximum value.
- *atmos.high_precip_low_temp* returns days with freezing rain conditions (low temperature and precipitations).
- *land.snow_cover_duration* computes the number of days snow depth exceeds some minimal threshold.
- *land.continuous_snow_cover_start* and *land.continuous_snow_cover_end* identify the day of the year when snow depth crosses a threshold for a given period of time.
- *days_with_snow*, counts days with snow between low and high thresholds, e.g. days with high amount of snow (*indice* and *indicator* available).
- *fire_season*, creates a fire season mask from temperature and, optionally, snow depth time-series.

13.18.3 New features and enhancements

- *generic.count_domain* counts values within low and high thresholds.
- *run_length.season* returns a dataset storing the start, end and length of a *season*.
- Fire Weather indices now support dask-backed data.
- Objects from the *xclim.sdba* submodule can be created from their string repr or from the dataset they created.
- Fire Weather Index submodule replicates the R code of *cffdrs*, including fire season determination and overwintering of the drought_code.
- New *run_bounds* and *keep_longest_run* utilities in *xclim.indices.run_length*.
- New bias-adjustment method: *PrincipalComponent* (based on Hnilica et al. 2017 https://doi.org/10.1002/joc.4890).

13.18.4 Internal changes

- Small changes in the output of *indices.run_length.rle*.

13.19 v0.24.0 (2021-03-01)

13.19.1 New indicators

- *days_over_precip_thresh*, *fraction_over_precip_thresh*, *liquid_precip_ratio*, *warm_spell_duration_index*, all from eponymous indices.
- *maximum_consecutive_warm_days* from indice *maximum_consecutive_tx_days*.

13.19.2 Breaking changes

- Numerous changes to *xclim.core.calendar.percentile_doy*:
  - *per* now accepts a sequence as well as a scalar and as such the output has a percentiles axis.
  - *per* argument is now expected to between 0-100 (not 0-1).
  - input data must have a daily (or coarser) time frequency.
- Change in unit handling paradigm for indices, which as a result will lead to some indices returning values with different units. Note that related *Indicator* objects remain unchanged and will return units consistent with CF Convention. If you are concerned with code stability, please use *Indicator* objects. The change was necessary to resolve inconsistencies with xarray's *keep_attrs=True* context.
  - Indice functions now return output units that preserve consistency with input units. That is, feeding inputs in Celsius will yield outputs in Celsius instead of casting to Kelvin. In all cases the dimensionality is preserved.
  - Indice functions now accept non-daily data, but daily frequency is assumed by default if the frequency cannot be inferred.
- Removed the explicitly-installed *netCDF4* python library from the base installation, as this is never explicitly used (now only installed in the *docs* recipe for sdba documented example).
- Removed *xclim.core.checks*, which was deprecated since v0.18.

13.19.3 New features and enhancements

- Indicator now have docstrings generated from their metadata.
- Units and fixed choices set are parsed from indice docstrings into *Indicator.parameters*.
- Units of indices using the *declare_units* decorator are stored in *indice.in_units* and *indice.out_units*.
- Changes to *Indicator.format* and *Indicator.json* to ensure the resulting json really is serializable.

13.19.4 Internal changes

- Leave *missing_options* undefined in *land.fit* indicator to allow control via *set_options*.
- Modified *xclim.core.calendar.percentile_doy* to improve performance.
- New *xclim.core.calendar.compare_offsets* for comparing offset strings.
- New *xclim.indices.generic.get_op* to retrieve a function from a string representation of that operator.
- The CI pipeline has been migrated from Travis CI to GitHub Actions. All stages are still built using *tox*.
- Indice functions must always set the units (the *declare_units* decorator does no check anymore).
- New *xclim.core.units.rate2amout* to convert rates like precipitation to amounts.
- *xclim.core.units.pint2cfunits* now removes ' * ' symbols and changes ° to *delta_deg*.
- New *xclim.core.units.to_agg_units* and *xclim.core.units.infer_sampling_units* for unit handling involving aggregation operations along the time dimension.
- Added an indicators API page to the docs and links to there from the *Climate Indicators* page.

13.19.5 Bug fixes

- The unit handling change resolved a bug that prevented the use of *xr.set_options(keep_attrs=True)* with indices.

13.20 v0.23.0 (2021-01-22)

13.20.1 Breaking changes

- Renamed indicator *atmos.degree_days_depassment_date* to *atmos.degree_days_exceedance_date*.
- In *degree_days_exceedance_date* : renamed argument *start_date* to *after_date*.
- Added cfchecks for Pr+Tas-based indicators.
- Refactored test suite to now be available as part of the standard library installation (*xclim.testing.tests*).
- Running *pytest* with *xdoctest* now requires the *rootdir* to point at *tests* location (*pytest –rootdir xclim/testing/tests/ –xdoctest xclim*).
- Development checks now require working jupyter notebooks (assessed via the *pytest –nbval* command).

13.20.2 New indicators

- *rain_approximation* and *snowfall_approximation* for computing *prlp* and *prsn* from *pr* and *tas* (or *tasmin* or *tasmax*) according to some threshold and method.
- *solid_precip_accumulation* and *liquid_precip_accumulation* now accept a *thresh* parameter to control the binary snow/rain temperature threshold.
- *first_snowfall* and *last_snowfall* to compute the date of first/last snowfall exceeding a threshold in a period.

13.20.3 New features and enhancements

- New *kind* entry in the *parameters* property of indicators, differentiating between [optional] variables and parameters.
- The git pre-commit hooks (*pre-commit run –all*) now clean the jupyter notebooks with *nbstripout* call.

13.20.4 Bug fixes

- Fixed a bug in *indices.run_length.lazy_indexing* that occurred with 1D coords and 0D indexes when using the dask backend.
- Fixed a bug with default frequency handling affecting *fit* indicator.
- Set missing method to 'skip' for *freq_analysis* indicator.
- Fixed a bug in *ensembles._ens_align_datasets* that occurred when inputs are *.nc* filepaths but files lack a time dimension.

13.20.5 Internal changes

- *core.cfchecks.check_valid* now accepts a sequence of strings as its *expected* argument.
- Clean up in the tests to speed up testing. Addition of a marker to include "slow" tests when desired (*-m slow*).
- Fixes in the tests to support *sklearn>=0.24*, *clisops>=0.5* and build xarray@master against python 3.7.
- Moved the testing suite to within xclim and simplified *tox* to manage its own tempdir.
- Indicator class now has a *default_freq* method.

13.21 v0.22.0 (2020-12-07)

13.21.1 Breaking changes

- Statistical functions (*frequency_analysis*, *fa*, *fit*, *parametric_quantile*) are now solely accessible via *indices.stats*.

13.21.2 New indicators

- *atmos.degree_days_depassment_date*, the day of year when the degree days sum exceeds a threshold.

13.21.3 New features and enhancements

- Added unique titles to *atmos* calculations employing wrapped_partials.
- *xclim.core.calendar.convert_calendar* now accepts a *missing* argument.
- Added *xclim.core.calendar.date_range* and *xclim.core.calendar.date_range_like* wrapping pandas' *date_range* and xarray's *cftime_range*.
- *xclim.core.calendar.get_calendar* now accepts many different types of data, including datetime object directly.
- New module *xclim.analog* and method *xclim.analog.spatial_analogs* to compute spatial analogs.
- Indicators can now accept dataset in their new *ds* call argument. Variable arguments (that use the *DataArray* annotation) can now be given with strings that correspond to variable names in the dataset, and default to their own name.
- Clarification to *frequency_analysis* notebook.
- Now officially supporting PEP596 (Python3.9).
- New methods *xclim.ensembles.change_significance* and *xclim.ensembles.knutti_sedlacek* to qualify climate change agreement among members of an ensemble.

13.21.4 Bug fixes

- Fixed bug that prevented the use of *xclim.core.missing.MissingBase* and subclasses with an indexer and a cftime datetime coordinate.
- Fixed issues with metadata handling in statistical indices.
- Various small fixes to the documentation (re-establishment of some internally and externally linked documents).

13.21.5 Internal changes

- Passing *align_on* to *xclim.core.calendar.convert_calendar* without using '360_day' calendars will not raise a warning anymore.
- Added formatting utilities for metadata attributes (*update_cell_methods*, *prefix_attrs* and *unprefix_attrs*).
- *xclim/ensembles.py* moved to *xclim/ensembles/*.py*, splitting stats/creation, reduction and robustness methods.
- With the help of the *mypy* library, added several typing fixes to better identify inputs/outputs, and reduce object type mutations.
- Fixed some doctests in *ensembles* and *set_options*.
- *clisops* v0.4.0+ is now an optional requirements for non-Windows builds.
- New *xclim.core.units.str2pint* method to convert quantity strings to quantity objects. Main improvement is to make "3 degC days" a valid string that converts to "3 K days".

13.22 v0.21.0 (2020-10-23)

13.22.1 Breaking changes

- Statistical functions (*frequency_analysis*, *fa*, *fit*, *parametric_quantile*) moved from *indices.generic* to *indices.stats* to make them more visible.

13.22.2 New indicators

13.22.3 New features and enhancements

- New xclim.testing.open_dataset method to read data from the remote testdata repo.
- Added a notebook, *ensembles-advanced.ipynb*, to the documentation detailing ensemble reduction techniques and showing how to make use of built-in figure-generating commands.
- Added a notebook, *frequency_analysis.ipynb*, with examples showcasing frequency analysis capabilities.

13.22.4 Bug fixes

- Fixed a bug in the attributes of *frost_season_length*.
- *indices.run_length* methods using dates now respect the array's calendar.
- Worked around an xarray bug in sdba.QuantileDeltaMapping when multidimensional arrays are used with linear or cubic interpolation.

13.22.5 Internal changes

13.23 v0.20.0 (2020-09-18)

13.23.1 Breaking changes

- *xclim.subset* has been deprecated and now relies on *clisops* to perform specialized spatio-temporal subsetting. Install with *pip install xclim[gis]* in order to retain the same functionality.
- The python library *pandoc* is no longer listed as a docs build requirement. Documentation still requires a current version of *pandoc* binaries installed at system-level.
- ANUCLIM indices have seen their *input_freq* parameter renamed to *src_timestep* for clarity.
- A clean-up and harmonization of the indicators metadata has changed some of the indicator identifiers, long_names, abstracts and titles. *xclim.atmos.drought_code* and *fire_weather_indexes* now have indentifiers "dc" and "fwi" (lowercase version of the previous identifiers).
- *xc.indices.run_length.run_length_with_dates* becomes *xc.indices.run_length.season_length*. Its argument *date* is now optional and the default changes from "07-01" to *None*.
- *xc.indices.consecutive_frost_days* becomes *xc.indices.maximum_consecutive_frost_days*.
- Changed the *history* indicator output attribute to *xclim_history* in order to respect CF conventions.

13.23.2 New indicators

- *atmos.max_pr_intensity* acting on hourly data.
- *atmos.wind_vector_from_speed*, also the *wind_speed_from_vector* now also returns the "wind from direction".
- Richards-Baker flow flashiness indicator (*xclim.land.rb_flashiness_index*).
- *atmos.max_daily_temperature_range*.
- *atmos.cold_spell_frequency*.
- *atmos.tg_min* and *atmos.tg_max*.
- *atmos.frost_season_length*, *atmos.first_day_above*. Also, *atmos.consecutive_frost_days* now takes a *thresh* argument (default : 0 degC).

13.23.3 New features and enhancements

- *sdba.loess* submodule implementing LOESS smoothing tools used in *sdba.detrending.LoessDetrend*.
- xclim now depends on clisops for subsetting, offloading several heavy GIS dependencies. This improves maintainability and reduces the size of a "vanilla" xclim installation considerably.
- New *generic.parametric_quantile* function taking parameters estimated by *generic.fit* as an input.
- Add support for using probability weighted moments method in *generic.fit* function. Requires the *lmoments3* package, which is not included in dependencies because it is unmaintained. Install manually if needed.
- Implemented *_fit_start* utility function providing initial conditions for statistical distribution parameters estimation, reducing the likelihood of poor fits.
- Added support for indicators based on hourly (1H) inputs, and a first hourly indicator called *max_pr_intensity* returning hourly precipitation intensity.
- Indicator instances can be retrieved through their class with the *get_instance()* class method. This allows the use of *xclim.core.indicator.registry* as an instance registry.
- Indicators now have a *realm* attribute. It must be given when creating indicators outside xclim.
- Better docstring parsing for indicators: parameters description, annotation and default value are accessible in the json output and *Indicator.parameters*.
- New command line interface *xclim* for simple indicator computing tasks.
- New *sdba.processing.jitter_over_thresh* for variables with a upper bound.
- Added *op* parameter to *xclim.indices.daily_temperature_range* to allow resample reduce operations other than mean
- *core.formatting.AttrFormatter* (and thus, locale dictionaries) can now use glob-like pattern for matching values to translate.

13.23.4 Bug fixes

The ICCLIM module was identified as *icclim* in the documentation but the module available under *ICCLIM*. Now *icclim* == *ICCLIM* and *ICCLIM will be deprecated in a future release.*

13.23.5 Internal changes

- *xclim.subset* now attempts to load and expose the functions of *clisops.core.subset*. This is an API workaround preserving backwards compatibility.
- Code styling now conforms to the latest release of black (v0.20.8).
- New *IndicatorRegistrar* class that takes care of adding indicator classes and instances to the appropriate registries. *Indicator* now inherits from it.

13.24 v0.19.0 (2020-08-18)

13.24.1 Breaking changes

- Refactoring of the *Indicator* class. The *cfprobe* method has been renamed to *cfcheck* and the *validate* method has been renamed to *datacheck*. More importantly, instantiating *Indicator* creates a new subclass on the fly and stores it in a registry, allowing users to subclass existing indicators easily. The algorithm for missing values is identified by its registered name, e.g. "any", "pct", etc, along with its *missing_options*.
- xclim now requires xarray >= 0.16, ensuring that xclim.sdba is fully functional.
- The dev requirements now include *xdoctest* – a rewrite of the standard library module, *doctest*.
- *xclim.core.locales.get_local_attrs* now uses the indicator's class name instead of the indicator itself and no longer accepts the *fill_missing* keyword. Behaviour is now the same as passing *False*.
- *Indicator.cf_attrs* is now a list of dictionaries. *Indicator.json* puts all the metadata attributes in the key "outputs" (a list of dicts). All variable metadata (names in *Indicator._cf_names*) might be strings or lists of strings when accessed as object attributes.
- Passing doctests are now strictly enforced as a build requirement in the Travis CI testing ensemble.

13.24.2 New features and enhancements

- New *ensembles.kkz_reduce_ensemble* method to select subsets of an ensemble based on the KKZ algorithm.
- Create new Indicator *Daily*, *Daily2D* subclasses for indicators using daily input data.
- The *Indicator* class now supports outputting multiple indices for the same inputs.
- *xclim.core.units.declare_units* now works with indices outputting multiple DataArrays.
- Doctests now make use of the *xdoctest_namespace* in order to more easily access modules and testdata.

13.24.3 Bug fixes

- Fix *generic.fit* dimension ordering. This caused errors when "time" was not the first dimension in a DataArray.

13.24.4 Internal changes

- *datachecks.check_daily* now uses *xr.infer_freq*.
- Indicator subclasses *Tas*, *Tasmin*, *Tasmax*, *Pr* and *Streamflow* now inherit from *Daily*.
- Indicator subclasses *TasminTasmax* and *PrTas* now inherit from *Daily2D*.
- Docstring style now enforced using the *pydocstyle* with *numpy* docstring conventions.
- Doctests are now performed for all docstring *Examples* using *xdoctest*. Failing examples must be explicitly skipped otherwise build will now fail.
- Indicator methods *update_attrs* and *format* are now classmethods, attrs to update must be passed.
- Indicators definitions without an accompanying translation (presently French) will cause build failures.
- Major refactoring of the internal machinery of *Indicator* to support multiple outputs.

13.25 v0.18.0 (2020-06-26)

- Optimization options for *xclim.sdba* : different grouping for the normalization steps of DQM and save training or fitting datasets to temporary files.
- *xclim.sdba.detrending* objects can now act on groups.
- Replaced *dask[complete]* with *dask[array]* in basic installation and added *distributed* to *docs* build dependencies.
- *xclim.core.locales* now supported in Windows build environments.
- *ensembles.ensemble_percentiles* modified to compute along a *percentiles* dimension by default, instead of creating different variables.
- Added indicator *first_day_below* and run length helper *first_run_after_date*.
- Added ANUCLIM model climate indices mappings.
- Renamed *areacella* to *areacello* in sea ice tests.
- Sea ice extent and area outputs now have units of m2 to comply with CF-Convention.
- Split *checks.py* into *cfchecks.py*, *datachecks.py* and *missing.py*. This change will only affect users creating custom indices using utilities previously located in *checks.py*.
- Changed signature of *daily_freeze_thaw_cycles*, *daily_temperature_range*, *daily_temperature_range_variability* and *extreme_temperature_range* to take (tasmin, tasmax) instead of (tasmax, tasmin) and match signature of other similar multivariate indices.
- Added *FromContext* subclass of *MissingBase* to have a uniform API for missing value operations.
- Remove logging commands that captured all xclim warnings. Remove deprecated xr.set_options calls.

13.26 v0.17.0 (2020-05-15)

- Added support for operations on dimensionless variables (*units = '1'*).
- Moved *xclim.locales* to *xclim.core.locales* in a batch of internal changes aimed to removed most potential cyclic imports cases.
- Missing checks and input validation refactored with addition of custom missing class registration (*xclim.core.checks.register_missing_method*) and simple validation method decorator (*xclim.core.checks.check*).
- New *xclim.set_options* context to control the missing checks, input validation and locales.
- New *xclim.sdba* module for statistical downscaling and bias-adjustment of climate data.
- Added *convert_calendar* and *interp_calendar* to help in the conversion between calendars.
- Added *at_least_n_valid* function, identifying null calculations based on minimum threshold.
- Added support for *freq=None* in missing calculations.
- Fixed outdated code examples in the docs and docstrings.
- Doctests are now run as part of the test suite.

13.27 v0.16.0 (2020-04-23)

- Added *vectorize* flag to *subset_shape* and *create_mask_vectorize* function based on *shapely.vectorize* as default backend for mask creation.
- Removed *start_yr* and *end_yr* flags from subsetting functions.
- Add multi gridpoints support in *subset.subset_gridpoint*.
- Better *wrapped_partial* for more meaningful inspection.
- Add indices for relative humidity, specific humidity and saturation vapor pressure with a few choices of method.
- Allow lazy units conversion.
- CRS definitions of projected DataSets are now written to file according to Climate and Forecast-convention standards.
- Add utilities to merge attributes and update history in xclim.core.formatting.
- Ensembles : Allow alignment of datasets with same frequency but different offsets.
- Bug fixes in run_length for run-with-dates methods when the date is not found in the run.
- Remove deepcopy from subset.subset_shape to improve memory usage.
- Add *missing_wmo* function, identifying null calculations based on criteria from WMO.
- Add *missing_pct* function, identifying null calculations based on percentage of missing values.

13.28 v0.15.x (2020-03-12)

- Improvement in FWI: Vectorization of DC, DMC and FFMC with numba and small code refactoring for better maintainability.
- Added example notebook for creating a catalog of selected indices
- Added *growing_season_end*, *last_spring_frost*, *dry_days*, *hot_spell_frequency*, *hot_spell_max_length*, and *maximum_consecutive_frost_free_days* indices.
- Dropped use of *fiona.crs* class in lieu of the newer pyproj CRS handler for *subset_shape* operations.
- Complete internal reorganization of xclim.
- Internationalization of xclim : add *locales* submodule for localized metadata.
- Add feature to retrieve coordinate values instead of index in *run_length.first_run*. Add *run_length.last_run*.
- Fix bug in subset_gridpoint to work on lat/lon coords of any dimension when they are not a dimension of the data.

13.29 v0.14.x (2020-02-21)

- Refactoring of the documentation.
- Added support for pint 0.10
- Add *atmos.heat_wave_total_length* (fixing a namespace issue)
- Fixes in *utils.percentile_doy* and *indices.winter_rain_ratio* for multidimensionnal datasets.
- Rewrote the *subset.subset_shape* function to allow for dask.delayed (lazy) computation.
- Added utility functions to compute *time_bnds* when resampling data encoded with *CFTimeIndex* (non-standard calendars).
- Fix in *subset.subset_gridpoint* for dask array coordinates.
- Modified *subset_shape* to support subsetting with GeoPandas datatypes directly.
- Fix in *subset.wrap_lons_and_split_at_greenwich* to preserve multi-region dataframes.
- Improve the memory use of *indices.growing_season_length*.
- Better handling of data with atypically named *lat* and *lon* dimensions.
- Added six Fire Weather indices.

13.30 v0.13.x (2020-01-10)

- Documentation improvements: list of indicators, RTD theme, notebook example.
- Added *sea_ice_extent* and *sea_ice_area* indicators.
- Reverted #311, removing the *_rolling* util function. Added optimal keywords to *rolling()* calls.
- Fixed *ensembles.create_ensemble* errors for builds against xarray master branch.
- Reformatted code to make better use of Python3.6 conventions (f-strings and object signatures).

- Fixed randomly failing tests of *checks.missing_any*.
- Improvement of *ensemble.ensemble_percentile* and *ensemble.create_ensemble*.

13.31 v0.12.x-beta (2019-11-18)

- Added a distance function computing the geodesic distance to a point.
- Added a *tolerance* argument to *subset_gridpoint* raising an error if distance to closest point is larger than tolerance.
- Created land module for standardized access to streamflow indices.
- Enhancement to utils.Indicator to have more dynamic attributes using callables.
- Added indices *heat_wave_total_length* and *tas / tg* to average tasmin and tasmax into tas.
- Fixed a bug with typed call signatures that caused downstream failures on library import.
- Added a *_rolling* util function to fix memory issues on large dask datasets.
- Added the *subset_shape* function to subset utilities for clipping region-masked datasets via polygons.
- Fixed a bug where certain dependencies caused ReadTheDocs builds to fail.
- Added many statically typed function signatures for better function documentation.
- Improved *DeprecationWarnings* and *UserWarnings* ensemble for xclim subsetting functions.
- Dropped support for Python3.5.

13.32 v0.11.x-beta (2019-10-17)

- Added type hinting to call signatures of many functions for more explicit type-checking.
- Added Kmeans clustering ensemble reduction algorithms.
- Added utilities for converting between wind velocity (sfcWind) and wind components (uas, vas) arrays.
- Added type hinting to call signatures of many functions for more explicit type-checking.
- Now supporting explicit builds for Windows OS via Travis CI.
- Fix failing test with Python 3.7.
- Fixed bug in subset.subset_bbox that could add unwanted coordinates/dims to some variables when applied to an entire dataset.
- Reformatted packaging configuration to pure Py3 wheel that ignore tests and test data.
- Now officially supporting Python3.8!
- Enhancement to precip_accumulation() to allow estimated amounts solid (or liquid) phase precipitation.
- Bugfix for frequency analysis choking on time series with NaNs only.

13.33 v0.10.x-beta (2019-06-18)

- Added indices to ICCLIM module.
- Added indices *days_over_precip_thresh* and *fraction_over_precip_thresh*.
- Migrated to a *major.minor.patch-release* semantic versioning system.
- Removed attributes in netCDF output from Indicators that are not in the CF-convention.
- Added *fit* indicator to fit the parameters of a distribution to a series.
- Added utilities with ensemble, run length, and subset algorithms to the documentation.
- Source code development standards now implement Python Black formatting.
- Pre-commit is now used to launch code formatting inspections for local development.
- Documentation now includes more detailed usage and an example workflow notebook.
- Development build configurations are now available via both Anaconda and pip install methods.
- Modified create_ensembles() to allow creation of ensemble dataset without a time dimension as well as from xr.Datasets.
- Modified create ensembles() to pad input data with nans when time dimensions are unequal.
- Updated subset_gridpoint() and subset_bbox() to use .sel method if 'lon' and 'lat' dims are present.
- *Added Azure Pipelines to automatically build xclim in Microsoft Windows environments.* – **REMOVED**
- Now employing PEP8 + Black compatible autoformatting.
- Added Windows and macOS images to Travis CI build ensemble.
- Added variable thresholds for tasmax and tasmin in daily_freezethaw_events.
- Updated subset.py to use date formatted strings ("%Y", "%Y%m" etc.) in temporal subsetting.
- Clean-up of day-of-year resampling. Precipitation percentile threshold will work without a doy index.
- Addressed deprecations for xarray 0.13.0.
- Added a decorator function that verifies validity and reformats subset calls using start_date or end_date signatures.
- Fixed a bug where 'lon' or 'lon_bounds' would return false values if either signatures were set to 0.

13.34 v0.10-beta (2019-06-06)

- Dropped support for Python 2.
- Added support for *period of the year* subsetting in `checks.missing_any`.
- Now allow for passing positive longitude values when subsetting data with negative longitudes.
- Improved runlength calculations for small grid size arrays via `ufunc_1dim` flag.

13.35 v0.9-beta (2019-05-13)

This is a significant jump in the release. Many modifications have been made and will be added to the documentation in the coming days. Among the many changes:

- New indices have been added with documentation and call examples.
- Run_length based operations have been optimized.
- Support for CF non-standard calendars.
- Automated/improved unit conversion and management via pint library.
- Added ensemble utilities for creation and analysis of muti-model climate ensembles.
- Added subsetting utilities for spatio-temporal subsets of xarray data objects.
- Added streamflow indicators.
- Refactoring of the code : separation of indices.py into a directory with sub-files (simple, threshold and multivariate); ensembles and subset utilities separated into distinct modules (pulled from utils.py).
- Indicators are now split into packages named by realms. import xclim.atmos to load indicators related to atmospheric variables.

13.36 v0.8-beta (2019-02-11)

This was a staging release and is functionally identical to 0.7-beta.

13.37 0.7-beta (2019-02-05)

Major Changes:

- Support for resampling of data structured using non-standard CF-Time calendars.
- Added several ICCLIM and other indicators.
- Dropped support for Python 3.4.
- Now under Apache v2.0 license.
- Stable PyPI-based dependencies.
- Dask optimizations for better memory management.
- Introduced class-based indicator calculations with data integrity verification and CF-Compliant-like metadata writing functionality.

Class-based indicators are new methods that allow index calculation with error-checking and provide on-the-fly metadata checks for CF-Compliant (and CF-compliant-like) data that are passed to them. When written to NetCDF, outputs of these indicators will append appropriate metadata based on the indicator, threshold values, moving window length, and time period / resampling frequency examined.

13.38 v0.6-alpha (2018-10-03)

- File attributes checks.
- Added daily downsampler function.
- Better documentation on ICCLIM indices.

13.39 v0.5-alpha (2018-09-26)

- Added total precipitation indicator.

13.40 v0.4-alpha (2018-09-14)

- Fully PEP8 compliant and available under MIT License.

13.41 v0.3-alpha (2018-09-4)

- Added icclim module.
- Reworked documentation, docs theme.

13.42 v0.2-alpha (2018-08-27)

- Added first indices.

13.43 v0.1.0-dev (2018-08-23)

- First release on PyPI.

CHAPTER

FOURTEEN

API

The API of the statistical downscaling and bias adjustment module (sdba) is documented *on this page*. The API of the `cfchecks`, `datachecks`, `missing` and `dataflags` modules are in *Health Checks*. Finally, the API of the translating tools is on the *Internationalization* page.

14.1 Indicators

14.1.1 Atmospheric indicators

While the *indices* module stores the computing functions, this module defines Indicator classes and instances that include a number of functionalities, such as input validation, unit conversion, output meta-data handling, and missing value masking.

The concept followed here is to define Indicator subclasses for each input variable, then create instances for each indicator.

`xclim.indicators.atmos.biologically_effective_degree_days(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', lat: Union[DataArray, str] = 'lat', *, thresh_tasmin: str = '10 degC', low_dtr: str = '10 degC', high_dtr: str = '13 degC', max_daily_degree_days: str = '9 degC', start_date: DayOfYearStr = '04-01', end_date: DayOfYearStr = '11-01', freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Biologically effective growing degree days. (realm: atmos)

Growing-degree days with a base of 10°C and an upper limit of 19°C and adjusted for latitudes between 40°N and 50°N for April to October (Northern Hemisphere; October to April in Southern Hemisphere). A temperature range adjustment also promotes small and large swings in daily temperature range. Used as a heat-summation metric in viticulture agroclimatology.

This indicator will check for missing values according to the method "from_context". Based on indice `biologically_effective_degree_days()`. With injected parameters: method=gladstones.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **lat** (*str or DataArray*) – Latitude coordinate. Default : *ds.lat*. [Required units : []]
- **thresh_tasmin** (*quantity (string with units)*) – The minimum temperature threshold. Default : 10 degC. [Required units : [temperature]]
- **low_dtr** (*quantity (string with units)*) – The lower bound for daily temperature range adjustment (default: 10°C). Default : 10 degC. [Required units : [temperature]]
- **high_dtr** (*quantity (string with units)*) – The higher bound for daily temperature range adjustment (default: 13°C). Default : 13 degC. [Required units : [temperature]]
- **max_daily_degree_days** (*quantity (string with units)*) – The maximum amount of biologically effective degrees days that can be summed daily. Default : 9 degC. [Required units : [temperature]]
- **start_date** (*date (string, MM-DD)*) – The hemisphere-based start date to consider (north = April, south = October). Default : 04-01.
- **end_date** (*date (string, MM-DD)*) – The hemisphere-based start date to consider (north = October, south = April). This date is non-inclusive. Default : 11-01.
- **freq** (*offset alias (string)*) – Resampling frequency (default: "YS"; For Southern Hemisphere, should be "AS-JUL"). Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

bedd (*DataArray*) – Biologically effective degree days computed with {method} formula (Summation of min((max((Tmin + Tmax)/2 - {thresh_tasmin}, 0) * k) + TR_adg, 9°C), for days between {start_date} and {end_date}). [K days] description: Heat-summation index for agroclimatic suitability estimation, developed specifically for viticulture. Considers daily Tmin and Tmax with a base of {thresh_tasmin} between 1 April and 31 October, with a maximum daily value for degree days (typically 9°C). It also integrates a modification coefficient for latitudes between 40°N and 50°N as well as swings in daily temperature range. comment: Original formula published in Gladstones, 1992.

Notes

The tasmax ceiling of 19°C is assumed to be the max temperature beyond which no further gains from daily temperature occur. Indice originally published in [Gladstones1992].

Let TX_i and TN_i be the daily maximum and minimum temperature at day i, lat the latitude of the point of interest, $degdays_{max}$ the maximum amount of degrees that can be summed per day (typically, 9). Then the sum of daily biologically effective growing degree day (BEDD) units between 1 April and 31 October is:

$$BEDD_i = \sum_{i=\text{April }1}^{\text{October }31} min\left(\left(max\left(\frac{TX_i + TN_i)}{2} - 10, 0\right) * k\right) + TR_{adj}, degdays_{max}\right)$$

$$TR_{adj} = f(TX_i, TN_i) = \begin{cases} 0.25(TX_i - TN_i - 13), & \text{if } (TX_i - TN_i) > 13 \\ 0, & \text{if } 10 < (TX_i - TN_i) < 13 \\ 0.25(TX_i - TN_i - 10), & \text{if } (TX_i - TN_i) < 10 \end{cases}$$

$$k = f(lat) = 1 + \left(\frac{|lat|}{50} * 0.06, \text{if } 40 < |lat| < 50, \text{else } 0\right)$$

A second version of the BEDD (*method="icclim"*) does not consider TR_{adj} and k and employs a different end date (30 September) ([ECAD]). The simplified formula is as follows:

$$BEDD_i = \sum_{i=\text{April 1}}^{\text{September 30}} min\left(max\left(\frac{TX_i + TN_i)}{2} - 10, 0\right), degdays_{max}\right)$$

References

`xclim.indicators.atmos.calm_days`(*sfcWind: Union[DataArray, str] = 'sfcWind', *, thresh: str = '2 m s-1', freq: str = 'MS', ds: Dataset = None, **indexer*) → DataArray

Calm days. (realm: atmos)

The number of days with average near-surface wind speed below threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *calm_days()*.

Parameters

- **sfcWind** (*str or DataArray*) – Daily windspeed. Default : *ds.sfcWind*. [Required units : [speed]]
- **thresh** (*quantity (string with units)*) – Threshold average near-surface wind speed on which to base evaluation. Default : 2 m s-1. [Required units : [speed]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

calm_days (*DataArray*) – Number of days with surface wind speed below threshold (number_of_days_with_sfcWind_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with surface wind speed < {thresh}

Notes

Let WS_{ij} be the windspeed at day i of period j. Then counted is the number of days where:

$$WS_{ij} < Threshold[ms - 1]$$

`xclim.indicators.atmos.cold_and_dry_days`(*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', tas_per: Union[DataArray, str] = 'tas_per', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Cold and dry days (realm: atmos)

Returns the total number of days where "Cold" and "Dry" conditions coincide.

This indicator will check for missing values according to the method "from_context". Based on indice *cold_and_dry_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature values Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **tas_per** (*str or DataArray*) – First quartile of daily mean temperature computed by month. Default : *ds.tas_per*. [Required units : [temperature]]
- **pr_per** (*str or DataArray*) – First quartile of daily total precipitation computed by month. .. warning:: Before computing the percentiles, all the precipitation below 1mm must be filtered out ! Otherwise, the percentiles will include non-wet days. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

cold_and_dry_days (*DataArray*) – Cold and dry days [days] cell_methods: time: sum over days description: {freq} number of days where tas < {tas_per_thresh}th percentile and pr < {pr_per_thresh}th percentile

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [cold_dry_days].

References

`xclim.indicators.atmos.cold_and_wet_days`(*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', tas_per: Union[DataArray, str] = 'tas_per', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

cold and wet days (realm: atmos)

Returns the total number of days where "cold" and "wet" conditions coincide.

This indicator will check for missing values according to the method "from_context". Based on indice *cold_and_wet_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature values Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **tas_per** (*str or DataArray*) – First quartile of daily mean temperature computed by month. Default : *ds.tas_per*. [Required units : [temperature]]

- **pr_per** (*str or DataArray*) – Third quartile of daily total precipitation computed by month. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

cold_and_wet_days (*DataArray*) – cold and wet days [days] cell_methods: time: sum over days description: {freq} number of days where tas < {tas_per_thresh}th percentile and pr > {pr_per_thresh}th percentile

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [cold_wet_days].

References

`xclim.indicators.atmos.cold_spell_days(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '-10 degC', window: int = 5, freq: str = 'AS-JUL', ds: Dataset = None*`)` → DataArray

Cold spell days. (realm: atmos)

The number of days that are part of cold spell events, defined as a sequence of consecutive days with mean daily temperature below a threshold in °C.

This indicator will check for missing values according to the method "from_context". Based on indice *`cold_spell_days()`*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature below which a cold spell begins. Default : -10 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature below threshold to qualify as a cold spell. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

cold_spell_days (*DataArray*) – Number of days part of a cold spell (cold_spell_days) [days] description: {freq} number of days that are part of a cold spell, defined as {window} or more consecutive days with mean daily temperature below {thresh}.

Notes

Let T_i be the mean daily temperature on day i, the number of cold spell days during period ϕ is given by

$$\sum_{i\in\phi} \prod_{j=i}^{i+5} [T_j < thresh]$$

where $[P]$ is 1 if P is true, and 0 if false.

`xclim.indicators.atmos.cold_spell_duration_index(`*tasmin: Union[DataArray, str] = 'tasmin', tasmin_per: Union[DataArray, str] = 'tasmin_per', *, window: int = 6, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None*`)` → DataArray

Cold spell duration index. (realm: atmos)

Number of days with at least *window* consecutive days where the daily minimum temperature is below the *tasmin_per* percentiles.

This indicator will check for missing values according to the method "from_context". Based on indice *cold_spell_duration_index()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmin_per** (*str or DataArray*) – nth percentile of daily minimum temperature with *dayofyear* coordinate. Default : *ds.tasmin_per*. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature below threshold to qualify as a cold spell. Default : 6.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

csdi_{window} (*DataArray*) – Number of days part of a percentile-defined cold spell (cold_spell_duration_index) [days] description: {freq} number of days with at least {window} consecutive days where the daily minimum temperature is below the {tasmin_per_thresh}th percentile(s). A {tasmin_per_window} day(s) window, centred on each calendar day in the {tasmin_per_period} period, is used to compute the {tasmin_per_thresh}th percentile(s).

Notes

Let TN_i be the minimum daily temperature for the day of the year i and $TN10_i$ the 10th percentile of the minimum daily temperature over the 1961-1990 period for day of the year i, the cold spell duration index over period ϕ is defined as:

$$\sum_{i\in\phi}\prod_{j=i}^{i+6}[TN_j < TN10_j]$$

where $[P]$ is 1 if P is true, and 0 if false.

References

From the Expert Team on Climate Change Detection, Monitoring and Indices (ETCCDMI).

`xclim.indicators.atmos.cold_spell_frequency`(*tas: Union[DataArray, str] = 'tas', *, thresh: str = '-10 degC', window: int = 5, freq: str = 'AS-JUL', ds: Dataset = None*) → DataArray

Cold spell frequency. (realm: atmos)

The number of cold spell events, defined as a sequence of consecutive days with mean daily temperature below a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice `cold_spell_frequency()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature below which a cold spell begins. Default : -10 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature below threshold to qualify as a cold spell. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

cold_spell_frequency (*DataArray*) – Number of cold spell events (cold_spell_frequency) description: {freq} number cold spell events, defined as {window} or more consecutive days with mean daily temperature below {thresh}.

`xclim.indicators.atmos.consecutive_frost_days`(*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '0.0 degC', freq: str = 'AS-JUL', ds: Dataset = None*) → DataArray

Maximum number of consecutive frost days (Tn < 0°C). (realm: atmos)

The maximum number of consecutive days within the period where the temperature is under a certain threshold (default: 0°C). WARNING: The default freq value is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice `maximum_consecutive_frost_days()`.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature. Default : 0.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

consecutive_frost_days (*DataArray*) – Maximum number of consecutive days with Tmin < {thresh} (spell_length_of_days_with_air_temperature_below_threshold) [days] cell_methods: time: maximum over days description: {freq} maximum number of consecutive days with minimum daily temperature below {thresh}.

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily minimum temperature series and *thresh* the threshold below which a day is considered a frost day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i < thresh] \neq [t_{i+1} < thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive frost free days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indicators.atmos.cool_night_index(`*tasmin: Union[DataArray, str] = 'tasmin', lat: Union[DataArray, str] = 'lat', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Cool Night Index. (realm: atmos)

A night coolness variable which takes into account the mean minimum night temperatures during the month when ripening usually occurs beyond the ripening period.

This indicator will check for missing values according to the method "from_context". Based on indice `cool_night_index()`.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **lat** (*str or DataArray*) – Latitude coordinate. Default : *ds.lat*. [Required units : []]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

cool_night_index (*DataArray*) – cool night index [degC] cell_methods: time: mean over days description: Mean minimum temperature for September (northern hemisphere) or March (southern hemisphere).

Notes

Given that this indice only examines September and March months, it is possible to send in DataArrays containing only these timesteps. Users should be aware that due to the missing values checks in wrapped Indicators, datasets that are missing several months will be flagged as invalid. This check can be ignored by setting the following context:

References

`xclim.indicators.atmos.cooling_degree_days`(*tas: Union[DataArray, str] = 'tas', *, thresh: str = '18.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Cooling degree days. (realm: atmos)

Sum of degree days above the temperature threshold at which spaces are cooled.

This indicator will check for missing values according to the method "from_context". Based on indice *cooling_degree_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Temperature threshold above which air is cooled. Default : 18.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

cooling_degree_days (*DataArray*) – Cooling degree days (Tmean > {thresh}) (integral_of_air_temperature_excess_wrt_time) [K days] cell_methods: time: sum over days description: {freq} cooling degree days above {thresh}.

Notes

Let x_i be the daily mean temperature at day i. Then the cooling degree days above temperature threshold *thresh* over period ϕ is given by:

$$\sum_{i\in\phi}(x_i - thresh[x_i > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false.

`xclim.indicators.atmos.corn_heat_units`(*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '4.44 degC', thresh_tasmax: str = '10 degC', ds: Dataset = None*) → DataArray

Corn heat units. (realm: atmos)

Temperature-based index used to estimate the development of corn crops. Formula adapted from [BootsmaTremblay&Filion1999]_.

This indicator will check for missing values according to the method "skip". Based on indice `corn_heat_units()`.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The minimum temperature threshold needed for corn growth. Default : 4.44 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The maximum temperature threshold needed for corn growth. Default : 10 degC. [Required units : [temperature]]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

chu (*DataArray*) – Corn heat units (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax}). description: Temperature-based index used to estimate the development of corn crops. Corn growth occurs when the minimum and maximum daily temperature both exceeds specific thresholds : Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax}.

Notes

Formula used in calculating the Corn Heat Units for the Agroclimatic Atlas of Quebec [Audet&al2012]_.

The thresholds of 4.44°C for minimum temperatures and 10°C for maximum temperatures were selected following the assumption that no growth occurs below these values.

Let TX_i and TN_i be the daily maximum and minimum temperature at day i. Then the daily corn heat unit is:

$$CHU_i = \frac{YX_i + YN_i}{2}$$

with

$$\begin{aligned} YX_i &= 3.33(TX_i - 10) - 0.084(TX_i - 10)^2, && \text{if } TX_i > 10C \\ YN_i &= 1.8(TN_i - 4.44), && \text{if } TN_i > 4.44C \end{aligned}$$

where YX_i and YN_i is 0 when $TX_i \leq 10C$ and $TN_i \leq 4.44C$, respectively.

References

`xclim.indicators.atmos.daily_freezethaw_cycles(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '0 degC', thresh_tasmax: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Statistics of consecutive diurnal temperature swing events. (realm: atmos)

A diurnal swing of max and min temperature event is when Tmax > thresh_tasmax and Tmin <= thresh_tasmin. This indice finds all days that constitute these events and computes statistics over the length and frequency of these events.

This indicator will check for missing values according to the method "from_context". Based on indice *multiday_temperature_swing()*. With injected parameters: window=1, op=sum.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The temperature threshold needed to trigger a freeze event. Default : 0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The temperature threshold needed to trigger a thaw event. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

dlyfrzthw (*DataArray*) – daily freezethaw cycles [days] description: {freq} number of days with a diurnal freeze-thaw cycle : Tmax > {thresh_tasmax} and Tmin <= {thresh_tasmin}.

Notes

Let TX_i be the maximum temperature at day i and TN_i be the daily minimum temperature at day i. Then freeze thaw spells during a given period are consecutive days where:

$$TX_i > 0 \wedge TN_i < 0$$

This indice returns a given statistic of the found lengths, optionally dropping those shorter than the *window* argument. For example, *window=1* and *op='sum'* returns the same value as `daily_freezethaw_cycles()`.

`xclim.indicators.atmos.daily_pr_intensity(`*pr: Union[DataArray, str] = 'pr', *, thresh: str = '1 mm/day', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Average daily precipitation intensity. (realm: atmos)

Return the average precipitation over wet days.

This indicator will check for missing values according to the method "from_context". Based on indice *daily_pr_intensity()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
sdii (*DataArray*) – Average precipitation during wet days (SDII) (lwe_thickness_of_precipitation_amount) [mm/day] description: {freq} Simple Daily Intensity Index (SDII) : {freq} average precipitation for days with daily precipitation over {thresh}. This indicator is also known as the 'Simple Daily Intensity Index' (SDII).

Notes

Let $\mathbf{p} = p_0, p_1, \ldots, p_n$ be the daily precipitation and *thresh* be the precipitation threshold defining wet days. Then the daily precipitation intensity is defined as

$$\frac{\sum_{i=0}^{n} p_i [p_i \leq thresh]}{\sum_{i=0}^{n} [p_i \leq thresh]}$$

where $[P]$ is 1 if P is true, and 0 if false.

`xclim.indicators.atmos.daily_temperature_range(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Mean of daily temperature range. (realm: atmos)

The mean difference between the daily maximum temperature and the daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *daily_temperature_range()*. With injected parameters: op=mean.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
dtr (*DataArray*) – Mean Diurnal Temperature Range (air_temperature) [K] cell_methods: time range within days time: mean over days description: {freq} mean diurnal temperature range.

Notes

For a default calculation using *op='mean'* :

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the mean diurnal temperature range in period j is:

$$DTR_j = \frac{\sum_{i=1}^{I}(TX_{ij} - TN_{ij})}{I}$$

`xclim.indicators.atmos.daily_temperature_range_variability(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Mean absolute day-to-day variation in daily temperature range. (realm: atmos)

Mean absolute day-to-day variation in daily temperature range.

This indicator will check for missing values according to the method "from_context". Based on indice *daily_temperature_range_variability()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

dtrvar (*DataArray*) – Mean Diurnal Temperature Range Variability (air_temperature) [K] cell_methods: time range within days time: difference over days time: mean over days description: {freq} mean diurnal temparature range variability (defined as the average day-to-day variation in daily temperature range for the given time period)

Notes

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then calculated is the absolute day-to-day differences in period j is:

$$vDTR_j = \frac{\sum_{i=2}^{I} |(TX_{ij} - TN_{ij}) - (TX_{i-1,j} - TN_{i-1,j})|}{I}$$

`xclim.indicators.atmos.days_over_precip_doy_thresh(`*pr: Union[DataArray, str] = 'pr', pr_per: Union[DataArray, str] = 'pr_per', *, thresh: str = '1 mm/day', freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of wet days with daily precipitation over a given percentile. (realm: atmos)

Number of days over period where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *days_over_precip_thresh()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – Percentile of wet day precipitation flux. Either computed daily (one value per day of year) or computed over a period (one value per spatial point). Default : *ds.pr_per*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

days_over_precip_doy_thresh (*DataArray*) – Count of days with daily precipitation above the given percentile [days]. (number_of_days_with_lwe_thickness_of_precipitation_amount_above_daily_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with precipitation above the {pr_per_thresh}th daily percentile. Only days with at least {thresh} are counted. A {pr_per_window} day(s) window, centred on each calendar day in the {pr_per_period} period, is used to compute the {pr_per_thresh}th percentile(s).

`xclim.indicators.atmos.days_over_precip_thresh`(*pr: Union[DataArray, str] = 'pr', pr_per: Union[DataArray, str] = 'pr_per', *, thresh: str = '1 mm/day', freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*) → DataArray

Number of wet days with daily precipitation over a given percentile. (realm: atmos)

Number of days over period where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *days_over_precip_thresh()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – Percentile of wet day precipitation flux. Either computed daily (one value per day of year) or computed over a period (one value per spatial point). Default : *ds.pr_per*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

days_over_precip_thresh (*DataArray*) – Count of days with daily precipitation above the given percentile [days]. (number_of_days_with_lwe_thickness_of_precipitation_amount_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with precipitation above the {pr_per_thresh}th percentile of {pr_per_period} period. Only days with at least {thresh} are counted.

`xclim.indicators.atmos.days_with_snow`(*prsn: Union[DataArray, str] = 'prsn', *, low: str = '0 kg m-2 s-1', high: str = '1E6 kg m-2 s-1', freq: str = 'AS-JUL', ds: Dataset = None, **indexer*) → DataArray

Days with snowfall (realm: atmos)

Return the number of days where snowfall is within low and high thresholds.

This indicator will check for missing values according to the method "from_context". Based on indice *days_with_snow()*.

Parameters

- **prsn** (*str or DataArray*) – Solid precipitation flux. Default : *ds.prsn*. [Required units : [precipitation]]
- **low** (*quantity (string with units)*) – Minimum threshold solid precipitation flux. Default : 0 kg m-2 s-1. [Required units : [precipitation]]
- **high** (*quantity (string with units)*) – Maximum threshold solid precipitation flux. Default : 1E6 kg m-2 s-1. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

days_with_snow (*DataArray*) – Number of days with solid precipitation flux between low and high thresholds. [days] description: {freq} number of days with solid precipitation flux larger than {low} and smaller or equal to {high}.

References

Matthews, L., Andrey, J., & Picketts, I. (2017). Planning for Winter Road Maintenance in the Context of Climate Change, Weather, Climate, and Society, 9(3), 521-532, https://doi.org/10.1175/WCAS-D-16-0103.1

`xclim.indicators.atmos.degree_days_exceedance_date(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', sum_thresh: str = '25 K days', op: str = '>', after_date: DayOfYearStr = None, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Degree days exceedance date. (realm: atmos)

Day of year when the sum of degree days exceeds a threshold. Degree days are computed above or below a given temperature threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *degree_days_exceedance_date()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base degree days evaluation. Default : 0 degC. [Required units : [temperature]]
- **sum_thresh** (*quantity (string with units)*) – Threshold of the degree days sum. Default : 25 K days. [Required units : K days]
- **op** (*{'<=', 'lt', '<', 'ge', '>=', 'gt', 'le', '>'}*) – If equivalent to '>', degree days are computed as *tas - thresh* and if equivalent to '<', they are computed as *thresh - tas*. Default : >.
- **after_date** (*date (string, MM-DD)*) – Date at which to start the cumulative sum. In "mm-dd" format, defaults to the start of the sampling period. Default : None.
- **freq** (*offset alias (string)*) – Resampling frequency. If *after_date* is given, *freq* should be annual. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

degree_days_exceedance_date (*DataArray*) – Day of year when cumulative degree days exceed {sum_thresh}. (day_of_year) description: Day of year when the integral of degree days (tmean {op} {thresh}) exceeds {sum_thresh}, the cumulative sum starts on {after_date}.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j, T is the reference threshold and ST is the sum threshold. Then, starting at day :math:i_0:, the degree days exceedance date is the first day k such that

$$\begin{cases} ST < \sum_{i=i_0}^{k} \max(TG_{ij} - T, 0) & \text{if } op \text{ is '>'} \\ ST < \sum_{i=i_0}^{k} \max(T - TG_{ij}, 0) & \text{if } op \text{ is '<'} \end{cases}$$

The resulting k is expressed as a day of year.

Cumulated degree days have numerous applications including plant and insect phenology. See https://en.wikipedia.org/wiki/Growing_degree-day for examples.

`xclim.indicators.atmos.drought_code(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', lat: Union[DataArray, str] = 'lat', snd: Optional[Union[DataArray, str]] = None, dc0: Optional[Union[DataArray, str]] = None, season_mask: Optional[Union[DataArray, str]] = None, *, season_method: str | None = None, overwintering: bool = False, dry_start: str | None = None, initial_start_up: bool = True, ds: Dataset = None, **params*`)` → DataArray

Drought code (FWI component). (realm: atmos)

The drought code is part of the Canadian Forest Fire Weather Index System. It is a numeric rating of the average moisture content of organic layers.

This indicator will check for missing values according to the method "skip". Based on indice *`drought_code()`*.

Parameters

- **tas** (*str or DataArray*) – Noon temperature. Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Rain fall in open over previous 24 hours, at noon. Default : *ds.pr*. [Required units : [precipitation]]
- **lat** (*str or DataArray*) – Latitude coordinate Default : *ds.lat*. [Required units : []]
- **snd** (*str or DataArray, optional*) – Noon snow depth. [Required units : [length]]
- **dc0** (*str or DataArray, optional*) – Initial values of the drought code. [Required units : []]
- **season_mask** (*str or DataArray, optional*) – Boolean mask, True where/when the fire season is active. [Required units : []]
- **season_method** (*{'LA08', None, 'GFWED', 'WF93'}*) – How to compute the start-up and shutdown of the fire season. If "None", no start-ups or shutdowns are computed, similar to the R fwi function. Ignored if *season_mask* is given. Default : None.
- **overwintering** (*boolean*) – Whether to activate DC overwintering or not. If True, either season_method or season_mask must be given. Default : False.
- **dry_start** (*{None, 'GFWED', 'CFS'}*) – Whether to activate the DC and DMC "dry start" mechanism and which method to use. , see `fire_weather_ufunc()`. Default : None.

- **initial_start_up** (*boolean*) – If True (default), grid points where the fire season is active on the first timestep go through a start_up phase for that time step. Otherwise, previous codes must be given as a continuing fire season is assumed for those points. Default : True.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **params** – Any other keyword parameters as defined in *xclim.indices.fwi.fire_weather_ufunc* and in `default_params`. Default : None.

Returns
dc (*DataArray*) – Drought Code (drought_code) description: Numeric rating of the average moisture content of organic layers.

Notes

See https://cwfis.cfs.nrcan.gc.ca/background/dsm/fwi, the module's doc and doc of `fire_weather_ufunc()` for more information.

References

Updated source code for calculating fire danger indexes in the Canadian Forest Fire Weather Index System, Y. Wang, K.R. Anderson, and R.M. Suddaby, INFORMATION REPORT NOR-X-424, 2015.

`xclim.indicators.atmos.dry_days`(*pr: Union[DataArray, str] = 'pr', *, thresh: str = '0.2 mm/d', freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Dry days. (realm: atmos)

The number of days with daily precipitation below threshold.

This indicator will check for missing values according to the method "from_context". Based on indice `dry_days()`.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0.2 mm/d. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
dry_days (*DataArray*) – Number of dry days (precip < {thresh}) (number_of_days_with_lwe_thickness_of_precipitation_amount_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with daily precipitation under {thresh}.

Notes

Let PR_{ij} be the daily precipitation at day i of period j. Then counted is the number of days where:

$$\sum PR_{ij} < Threshold[mm/day]$$

`xclim.indicators.atmos.dry_spell_frequency`(*pr: Union[DataArray, str] = 'pr', *, thresh: str = '1.0 mm', window: int = 3, freq: str = 'YS', op: str = 'sum', ds: Dataset = None*) → DataArray

Return the number of dry periods of n days and more. (realm: atmos)

Periods during which the accumulated or maximal daily precipitation amount on a window of n days is under threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *dry_spell_frequency()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation amount under which a period is considered dry. The value against which the threshold is compared depends on *op* . Default : 1.0 mm. [Required units : [length]]
- **window** (*number*) – Minimum length of the spells. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **op** (*{'sum', 'max'}*) – Operation to perform on the window. Default is "sum", which checks that the sum of accumulated precipitation over the whole window is less than the threshold. "max" checks that the maximal daily precipitation amount within the window is less than the threshold. This is the same as verifying that each individual day is below the threshold. Default : sum.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

dry_spell_frequency (*DataArray*) – The {freq} number of dry periods of minimum {window} days. description: The {freq} number of dry periods of {window} days and more, during which the {op} precipitation on a window of {window} days is under {thresh}.

`xclim.indicators.atmos.dry_spell_total_length`(*pr: Union[DataArray, str] = 'pr', *, thresh: str = '1.0 mm', window: int = 3, op: str = 'sum', freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Total length of dry spells. (realm: atmos)

Total number of days in dry periods of a minimum length, during which the maximum or accumulated precipitation within a window of the same length is under a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *dry_spell_total_length()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]

- **thresh** (*quantity (string with units)*) – Accumulated precipitation value under which a period is considered dry. Default : 1.0 mm. [Required units : [length]]
- **window** (*number*) – Number of days when the maximum or accumulated precipitation is under threshold. Default : 3.
- **op** (*{'sum', 'max'}*) – Reduce operation. Default : sum.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Indexing is done after finding the dry days, but before finding the spells. Default : None.

Returns

dry_spell_total_length (*DataArray*) – The {freq} total number of days in dry periods of minimum {window} days. [days] description: The {freq} number of days in dry periods of {window} days and more, during which the {op}precipitation within windows of {window} days is under {thresh}.

Notes

The algorithm assumes days before and after the timeseries are "wet", meaning that the condition for being considered part of a dry spell is stricter on the edges. For example, with *window=3* and *op='sum'*, the first day of the series is considered part of a dry spell only if the accumulated precipitation within the first 3 days is under the threshold. In comparison, a day in the middle of the series is considered part of a dry spell if any of the three 3-day periods of which it is part are considered dry (so a total of five days are included in the computation, compared to only 3.)

`xclim.indicators.atmos.extreme_temperature_range`(*tasmin: Union[DataArray, str] = 'tasmin'*, *tasmax: Union[DataArray, str] = 'tasmax'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Extreme intra-period temperature range. (realm: atmos)

The maximum of max temperature (TXx) minus the minimum of min temperature (TNn) for the given time period.

This indicator will check for missing values according to the method "from_context". Based on indice *extreme_temperature_range()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

etr (*DataArray*) – Intra-period Extreme Temperature Range (air_temperature) [K] description: {freq} range between the maximum of daily max temperature (tx_max) and the minimum of daily min temperature (tn_min)

Notes

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the extreme temperature range in period j is:

$$ETR_j = max(TX_{ij}) - min(TN_{ij})$$

`xclim.indicators.atmos.fire_season`(*tas: Union[DataArray, str] = 'tas', snd: Optional[Union[DataArray, str]] = None, *, method: str = 'WF93', freq: str | None = None, temp_start_thresh: str = '12 degC', temp_end_thresh: str = '5 degC', temp_condition_days: int = 3, snow_condition_days: int = 3, snow_thresh: str = '0.01 m', ds: Dataset = None*) → DataArray

Fire season mask. (realm: atmos)

Binary mask of the active fire season, defined by conditions on consecutive daily temperatures and, optionally, snow depths.

Based on indice *fire_season()*.

Parameters

- **tas** (*str or DataArray*) – Daily surface temperature, cffdrs recommends using maximum daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **snd** (*str or DataArray, optional*) – Snow depth, used with method == 'LA08'. [Required units : [length]]
- **method** (*{'LA08', 'GFWED', 'WF93'}*) – Which method to use. "LA08" and "GFWED" need the snow depth. Default : WF93.
- **freq** (*offset alias (string)*) – If given only the longest fire season for each period defined by this frequency, Every "seasons" are returned if None, including the short shoulder seasons. Default : None.
- **temp_start_thresh** (*quantity (string with units)*) – Minimal temperature needed to start the season. Default : 12 degC. [Required units : [temperature]]
- **temp_end_thresh** (*quantity (string with units)*) – Maximal temperature needed to end the season. Default : 5 degC. [Required units : [temperature]]
- **temp_condition_days** (*number*) – Number of days with temperature above or below the thresholds to trigger a start or an end of the fire season. Default : 3.
- **snow_condition_days** (*number*) – Parameters for the fire season determination. See `fire_season()`. Temperature is in degC, snow in m. The *snow_thresh* parameters is also used when *dry_start* is set to "GFWED". Default : 3.
- **snow_thresh** (*quantity (string with units)*) – Minimal snow depth level to end a fire season, only used with method "LA08". Default : 0.01 m. [Required units : [length]]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

fire_season (*DataArray*) – Fire season mask description: Fire season mask, computed with method {method}.

References

[Wotton&Flannigan1993]_

[Lawson&Armitage2008]_

`xclim.indicators.atmos.fire_weather_indexes`(*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', sfcWind: Union[DataArray, str] = 'sfcWind', hurs: Union[DataArray, str] = 'hurs', lat: Union[DataArray, str] = 'lat', snd: Optional[Union[DataArray, str]] = None, ffmc0: Optional[Union[DataArray, str]] = None, dmc0: Optional[Union[DataArray, str]] = None, dc0: Optional[Union[DataArray, str]] = None, season_mask: Optional[Union[DataArray, str]] = None, *, season_method: str | None = None, overwintering: bool = False, dry_start: str | None = None, initial_start_up: bool = True, ds: Dataset = None, **params*) → Tuple[DataArray, DataArray, DataArray, DataArray, DataArray, DataArray]

Fire weather indexes. (realm: atmos)

Computes the 6 fire weather indexes as defined by the Canadian Forest Service: the Drought Code, the Duff-Moisture Code, the Fine Fuel Moisture Code, the Initial Spread Index, the Build Up Index and the Fire Weather Index.

This indicator will check for missing values according to the method "skip". Based on indice `fire_weather_indexes()`.

Parameters

- **tas** (*str or DataArray*) – Noon temperature. Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Rain fall in open over previous 24 hours, at noon. Default : *ds.pr*. [Required units : [precipitation]]
- **sfcWind** (*str or DataArray*) – Noon wind speed. Default : *ds.sfcWind*. [Required units : [speed]]
- **hurs** (*str or DataArray*) – Noon relative humidity. Default : *ds.hurs*. [Required units : []]
- **lat** (*str or DataArray*) – Latitude coordinate Default : *ds.lat*. [Required units : []]
- **snd** (*str or DataArray, optional*) – Noon snow depth, only used if *season_method='LA08'* is passed. [Required units : [length]]
- **ffmc0** (*str or DataArray, optional*) – Initial values of the fine fuel moisture code. [Required units : []]
- **dmc0** (*str or DataArray, optional*) – Initial values of the Duff moisture code. [Required units : []]

- **dc0** (*str or DataArray, optional*) – Initial values of the drought code. [Required units : []]
- **season_mask** (*str or DataArray, optional*) – Boolean mask, True where/when the fire season is active. [Required units : []]
- **season_method** (*{'LA08', None, 'GFWED', 'WF93'}*) – How to compute the start-up and shutdown of the fire season. If "None", no start-ups or shutdowns are computed, similar to the R fwi function. Ignored if *season_mask* is given. Default : None.
- **overwintering** (*boolean*) – Whether to activate DC overwintering or not. If True, either season_method or season_mask must be given. Default : False.
- **dry_start** (*{None, 'GFWED', 'CFS'}*) – Whether to activate the DC and DMC "dry start" mechanism or not, see `fire_weather_ufunc()`. Default : None.
- **initial_start_up** (*boolean*) – If True (default), gridpoints where the fire season is active on the first timestep go through a start_up phase for that time step. Otherwise, previous codes must be given as a continuing fire season is assumed for those points. Default : True.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **params** – Any other keyword parameters as defined in `fire_weather_ufunc()` and in `default_params`. Default : None.

Returns

- **dc** (*DataArray*) – Drought Code (drought_code) description: Numeric rating of the average moisture content of deep, compact organic layers.
- **dmc** (*DataArray*) – Duff Moisture Code (duff_moisture_code) description: Numeric rating of the average moisture content of loosely compacted organic layers of moderate depth.
- **ffmc** (*DataArray*) – Fine Fuel Moisture Code (fine_fuel_moisture_code) description: Numeric rating of the average moisture content of litter and other cured fine fuels.
- **isi** (*DataArray*) – Initial Spread Index (initial_spread_index) description: Numeric rating of the expected rate of fire spread.
- **bui** (*DataArray*) – Buildup Index (buildup_index) description: Numeric rating of the total amount of fuel available for combustion.
- **fwi** (*DataArray*) – Fire Weather Index (fire_weather_index) description: Numeric rating of fire intensity.

Notes

See https://cwfis.cfs.nrcan.gc.ca/background/dsm/fwi, the module's doc and doc of `fire_weather_ufunc()` for more information.

References

Updated source code for calculating fire danger indexes in the Canadian Forest Fire Weather Index System, Y. Wang, K.R. Anderson, and R.M. Suddaby, INFORMATION REPORT NOR-X-424, 2015.

`xclim.indicators.atmos.first_day_above`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *thresh: str = '0 degC'*, *after_date: DayOfYearStr = '01-01'*, *window: int = 1*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

First day of temperatures superior to a threshold temperature. (realm: atmos)

Returns first day of period where a temperature is superior to a threshold over a given number of days, limited to a starting calendar date.

This indicator will check for missing values according to the method "from_context". Based on indice *first_day_above()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **after_date** (*date (string, MM-DD)*) – Date of the year after which to look for the first event. Should have the format '%m-%d'. Default : 01-01.
- **window** (*number*) – Minimum number of days with temperature above threshold needed for evaluation. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

first_day_above (*DataArray*) – First day of year with temperature above {thresh} (day_of_year) description: First day of year with temperature above {thresh} for at least {window} days.

`xclim.indicators.atmos.first_day_below`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *thresh: str = '0 degC'*, *after_date: DayOfYearStr = '07-01'*, *window: int = 1*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

First day of temperatures inferior to a threshold temperature. (realm: atmos)

Returns first day of period where a temperature is inferior to a threshold over a given number of days, limited to a starting calendar date.

This indicator will check for missing values according to the method "from_context". Based on indice *first_day_below()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **after_date** (*date (string, MM-DD)*) – Date of the year after which to look for the first frost event. Should have the format '%m-%d'. Default : 07-01.
- **window** (*number*) – Minimum number of days with temperature below threshold needed for evaluation. Default : 1.

- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
first_day_below (*DataArray*) – First day of year with temperature below {thresh} (day_of_year) description: First day of year with temperature below {thresh} for at least {window} days.

`xclim.indicators.atmos.first_snowfall`(*prsn: Union[DataArray, str] = 'prsn', *, thresh: str = '0.5 mm/day', freq: str = 'AS-JUL', ds: Dataset = None, **indexer*) → DataArray

First day with solid precipitation above a threshold. (realm: atmos)

Returns the first day of a period where the solid precipitation exceeds a threshold. WARNING: The default *freq* is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *first_snowfall()*.

Parameters

- **prsn** (*str or DataArray*) – Solid precipitation flux. Default : *ds.prsn*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold precipitation flux on which to base evaluation. Default : 0.5 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
first_snowfall (*DataArray*) – Date of first snowfall (day_of_year) description: {freq} first day where the solid precipitation flux exceeded {thresh}

References

Climate Projections for the National Capital Region (2020), Volume 1: Results and Interpretation for Key Climate Indices, Report 193600.00, Prepared for Ottawa by CBCL.

`xclim.indicators.atmos.fraction_over_precip_doy_thresh`(*pr: Union[DataArray, str] = 'pr', pr_per: Union[DataArray, str] = 'pr_per', *, thresh: str = '1 mm/day', freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*) → DataArray

Fraction of precipitation due to wet days with daily precipitation over a given percentile. (realm: atmos)

Percentage of the total precipitation over period occurring in days where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *fraction_over_precip_thresh()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – Percentile of wet day precipitation flux. Either computed daily (one value per day of year) or computed over a period (one value per spatial point). Default : *ds.pr_per*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

fraction_over_precip_doy_thresh (*DataArray*) – Fraction of precipitation over threshold during wet days. description: {freq} fraction of total precipitation due to days with precipitation above {pr_per_thresh}th daily percentile. Only days with at least {thresh} are included in the total. A {pr_per_window} day(s) window, centred on each calendar day in the {pr_per_period} period, is used to compute the {pr_per_thresh}th percentile(s).

`xclim.indicators.atmos.fraction_over_precip_thresh`(*pr: Union[DataArray, str] = 'pr'*, *pr_per: Union[DataArray, str] = 'pr_per'*, ***, *thresh: str = '1 mm/day'*, *freq: str = 'YS'*, *bootstrap: bool = False*, *ds: Dataset = None*, ***indexer*) → DataArray

Fraction of precipitation due to wet days with daily precipitation over a given percentile. (realm: atmos)

Percentage of the total precipitation over period occurring in days where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *fraction_over_precip_thresh()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – Percentile of wet day precipitation flux. Either computed daily (one value per day of year) or computed over a period (one value per spatial point). Default : *ds.pr_per*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.

- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

fraction_over_precip_thresh (*DataArray*) – Fraction of precipitation over threshold during wet days. description: {freq} fraction of total precipitation due to days with precipitation above {pr_per_thresh}th percentile of {pr_per_period} period. Only days with at least {thresh} are included in the total.

`xclim.indicators.atmos.freezethaw_spell_frequency`(*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '0 degC', thresh_tasmax: str = '0 degC', window: int = 1, freq: str = 'YS', ds: Dataset = None*) → DataArray

Frequency of freeze-thaw spells (realm: atmos)

A diurnal swing of max and min temperature event is when Tmax > thresh_tasmax and Tmin <= thresh_tasmin. This indice finds all days that constitute these events and computes statistics over the length and frequency of these events.

This indicator will check for missing values according to the method "from_context". Based on indice *multiday_temperature_swing()*. With injected parameters: op=count.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The temperature threshold needed to trigger a freeze event. Default : 0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The temperature threshold needed to trigger a thaw event. Default : 0 degC. [Required units : [temperature]]
- **window** (*number*) – The minimal length of spells to be included in the statistics. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

freezethaw_spell_frequency (*DataArray*) – {freq} number of freeze-thaw spells. [days] description: {freq} number of freeze-thaw spells: Tmax > {thresh_tasmax} and Tmin <= {thresh_tasmin} for at least {window} consecutive day(s).

Notes

Let TX_i be the maximum temperature at day i and TN_i be the daily minimum temperature at day i. Then freeze thaw spells during a given period are consecutive days where:

$$TX_i > 0 \land TN_i < 0$$

This indice returns a given statistic of the found lengths, optionally dropping those shorter than the *window* argument. For example, *window=1* and *op='sum'* returns the same value as `daily_freezethaw_cycles()`.

`xclim.indicators.atmos.freezethaw_spell_max_length(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '0 degC', thresh_tasmax: str = '0 degC', window: int = 1, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Maximal length of freeze-thaw spells. (realm: atmos)

A diurnal swing of max and min temperature event is when Tmax > thresh_tasmax and Tmin <= thresh_tasmin. This indice finds all days that constitute these events and computes statistics over the length and frequency of these events.

This indicator will check for missing values according to the method "from_context". Based on indice *multiday_temperature_swing()*. With injected parameters: op=max.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The temperature threshold needed to trigger a freeze event. Default : 0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The temperature threshold needed to trigger a thaw event. Default : 0 degC. [Required units : [temperature]]
- **window** (*number*) – The minimal length of spells to be included in the statistics. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

freezethaw_spell_max_length (*DataArray*) – {freq} maximal length of freeze-thaw spells. [days] description: {freq} maximal length of freeze-thaw spells: Tmax > {thresh_tasmax} and Tmin <= {thresh_tasmin} for at least {window} consecutive day(s).

Notes

Let TX_i be the maximum temperature at day i and TN_i be the daily minimum temperature at day i. Then freeze thaw spells during a given period are consecutive days where:

$$TX_i > 0 \land TN_i < 0$$

This indice returns a given statistic of the found lengths, optionally dropping those shorter than the *window* argument. For example, *window=1* and *op='sum'* returns the same value as `daily_freezethaw_cycles()`.

`xclim.indicators.atmos.freezethaw_spell_mean_length(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '0 degC', thresh_tasmax: str = '0 degC', window: int = 1, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Averge length of freeze-thaw spells. (realm: atmos)

A diurnal swing of max and min temperature event is when Tmax > thresh_tasmax and Tmin <= thresh_tasmin. This indice finds all days that constitute these events and computes statistics over the length and frequency of these events.

This indicator will check for missing values according to the method "from_context". Based on indice *multiday_temperature_swing()*. With injected parameters: op=mean.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The temperature threshold needed to trigger a freeze event. Default : 0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The temperature threshold needed to trigger a thaw event. Default : 0 degC. [Required units : [temperature]]
- **window** (*number*) – The minimal length of spells to be included in the statistics. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

freezethaw_spell_mean_length (*DataArray*) – {freq} average length of freeze-thaw spells. [days] description: {freq} average length of freeze-thaw spells: Tmax > {thresh_tasmax} and Tmin <= {thresh_tasmin} for at least {window} consecutive day(s).

Notes

Let TX_i be the maximum temperature at day i and TN_i be the daily minimum temperature at day i. Then freeze thaw spells during a given period are consecutive days where:

$$TX_i > 0 \wedge TN_i < 0$$

This indice returns a given statistic of the found lengths, optionally dropping those shorter than the *window* argument. For example, *window=1* and *op='sum'* returns the same value as `daily_freezethaw_cycles()`.

`xclim.indicators.atmos.freezing_degree_days`(*tas: Union[DataArray, str] = 'tas'*, ***, *thresh: str = '0 degC'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Heating degree days. (realm: atmos)

Sum of degree days below the temperature threshold at which spaces are heated.

This indicator will check for missing values according to the method "from_context". Based on indice *heating_degree_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

freezing_degree_days (*DataArray*) – Freezing degree days (Tmean < {thresh}) (integral_of_air_temperature_deficit_wrt_time) [K days] cell_methods: time: sum over days description: {freq} freezing degree days below {thresh}.

Notes

This index intentionally differs from its ECA&D equivalent: HD17. In HD17, values below zero are not clipped before the sum. The present definition should provide a better representation of the energy demand for heating buildings to the given threshold.

Let TG_{ij} be the daily mean temperature at day i of period j. Then the heating degree days are:

$$HD17_j = \sum_{i=1}^{I} (17 - TG_{ij}) | TG_{ij} < 17)$$

`xclim.indicators.atmos.freshet_start`(*tas: Union[DataArray, str] = 'tas'*, ***, *thresh: str = '0 degC'*, *window: int = 5*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

First day consistently exceeding threshold temperature. (realm: atmos)

Returns first day of period where a temperature threshold is exceeded over a given number of days.

This indicator will check for missing values according to the method "from_context". Based on indice *freshet_start()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold needed for evaluation. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

freshet_start (*DataArray*) – Day of year of spring freshet start (day_of_year) description: Day of year of spring freshet start, defined as the first day a temperature threshold of {thresh} is exceeded for at least {window} days.

Notes

Let x_i be the daily mean temperature at day of the year i for values of i going from 1 to 365 or 366. The start date of the freshet is given by the smallest index i for which

$$\prod_{j=i}^{i+w} [x_j > thresh]$$

is true, where w is the number of days the temperature threshold should be exceeded, and $[P]$ is 1 if P is true, and 0 if false.

`xclim.indicators.atmos.frost_days`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *thresh: str = '0 degC'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Frost days index. (realm: atmos)

Number of days where daily minimum temperatures are below a threshold temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *frost_days()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Freezing temperature. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

frost_days (*DataArray*) – Number of frost days (Tmin < {thresh}) (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with minimum daily temperature below {thresh}.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j and :math`TT` the threshold. Then counted is the number of days where:

$$TN_{ij} < TT$$

`xclim.indicators.atmos.frost_free_season_end(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '0 degC', mid_date: DayOfYearStr = '07-01', window: int = 5, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

End of the frost free season. (realm: atmos)

Day of the year of the start of a sequence of days with minimum temperatures consistently below a threshold, after a period with minimum temperatures consistently above the same threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *frost_free_season_end()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **mid_date** (*date (string, MM-DD)*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'. Default : 07-01.
- **window** (*number*) – Minimum number of days with temperature below threshold needed for evaluation. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

frost_free_season_end (*DataArray*) – Day of year of frost free season end (day_of_year) description: Day of year of end of frost free season, defined as the first day minimum temperatures below a threshold of {thresh}, after a run of days above this threshold, for at least {window} days.

`xclim.indicators.atmos.frost_free_season_length(`*tasmin: Union[DataArray, str] = 'tasmin', *, window: int = 5, mid_date: DayOfYearStr | None = '07-01', thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Frost free season length. (realm: atmos)

The number of days between the first occurrence of at least N (def: 5) consecutive days with minimum daily temperature above a threshold (default: 0℃) and the first occurrence of at least N (def 5) consecutive days with minimum daily temperature below the same threshold A mid date can be given to limit the earliest day the end of season can take. WARNING: The default freq and mid_date values are valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice `frost_free_season_length()`.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold to mark the beginning and end of frost free season. Default : 5.
- **mid_date** (*date (string, MM-DD)*) – Date the must be included in the season. It is the earliest the end of the season can be. If None, there is no limit. Default : 07-01.
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

frost_free_season_length (*DataArray*) – Length of the frost free season (days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days between the first occurrence of at least {window} consecutive days with minimum daily temperature above or at the freezing point and the first occurrence of at least {window} consecutive days with minimum daily temperature below freezing after {mid_date}.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then counted is the number of days between the first occurrence of at least N consecutive days with:

$$TN_{ij} >= 0$$

and the first subsequent occurrence of at least N consecutive days with:

$$TN_{ij} < 0$$

`xclim.indicators.atmos.frost_free_season_start(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '0 degC', window: int = 5, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Start of the frost free season. (realm: atmos)

Day of the year of the start of a sequence of days with minimum temperatures consistently above or equal to a threshold, after a period with minimum temperatures consistently above the same threshold.

This indicator will check for missing values according to the method "from_context". Based on indice `frost_free_season_start()`.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]

- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold needed for evaluation. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

frost_free_season_start (*DataArray*) – Day of year of frost free season start (day_of_year) description: Day of year of beginning of frost free season, defined as the first day a minimum temperature threshold of {thresh} is equal or exceeded for at least {window} days.

Notes

Let x_i be the daily mean temperature at day of the year i for values of i going from 1 to 365 or 366. The start date of the start of growing season is given by the smallest index i for which:

$$\prod_{j=i}^{i+w} [x_j >= thresh]$$

is true, where w is the number of days the temperature threshold should be met or exceeded, and $[P]$ is 1 if P is true, and 0 if false.

```
xclim.indicators.atmos.frost_season_length(tasmin: Union[DataArray, str] = 'tasmin', *, window:
                                           int = 5, mid_date: DayOfYearStr | None = '01-01',
                                           freq: str = 'AS-JUL', ds: Dataset = None) →
                                           DataArray
```

Frost season length. (realm: atmos)

The number of days between the first occurrence of at least N (def: 5) consecutive days with minimum daily temperature under a threshold (default: 0℃) and the first occurrence of at least N (def 5) consecutive days with minimum daily temperature above the same threshold A mid date can be given to limit the earliest day the end of season can take. WARNING: The default freq and mid_date values are valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *frost_season_length()*. With injected parameters: thresh=0 degC.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature below threshold to mark the beginning and end of frost season. Default : 5.
- **mid_date** (*date (string, MM-DD)*) – Date the must be included in the season. It is the earliest the end of the season can be. If None, there is no limit. Default : 01-01.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

frost_season_length (*DataArray*) – Length of the frost season (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days between the first occurrence of at least {window} consecutive days with minimum daily temperature below freezing and the first occurrence of at least {window} consecutive days with minimuim daily temperature above freezing after {mid_date}.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then counted is the number of days between the first occurrence of at least N consecutive days with:

$$TN_{ij} > 0$$

and the first subsequent occurrence of at least N consecutive days with:

$$TN_{ij} < 0$$

`xclim.indicators.atmos.growing_degree_days(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '4.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Growing degree-days over threshold temperature value. (realm: atmos)

The sum of degree-days over the threshold temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *growing_degree_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 4.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

growing_degree_days (*DataArray*) – Growing degree days above {thresh} (integral_of_air_temperature_excess_wrt_time) [K days] cell_methods: time: sum over days description: {freq} growing degree days above {thresh}.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then the growing degree days are:

$$GD4_j = \sum_{i=1}^{I} (TG_{ij} - 4 | TG_{ij} > 4)$$

`xclim.indicators.atmos.growing_season_end(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '5.0 degC', mid_date: DayOfYearStr = '07-01', window: int = 5, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

End of the growing season. (realm: atmos)

Day of the year of the start of a sequence of days with mean temperatures consistently below a threshold, after a period with mean temperatures consistently above the same threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *growing_season_end()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 5.0 degC. [Required units : [temperature]]
- **mid_date** (*date (string, MM-DD)*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'. Default : 07-01.
- **window** (*number*) – Minimum number of days with temperature below threshold needed for evaluation. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

growing_season_end (*DataArray*) – Day of year of growing season end (day_of_year) description: Day of year of end of growing season, defined as the first day of consistent inferior threshold temperature of {thresh} after a run of {window} days superior to threshold temperature.

`xclim.indicators.atmos.growing_season_length(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '5.0 degC', window: int = 6, mid_date: DayOfYearStr = '07-01', freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Growing season length. (realm: atmos)

The number of days between the first occurrence of at least six consecutive days with mean daily temperature over a threshold (default: 5℃) and the first occurrence of at least six consecutive days with mean daily temperature below the same threshold after a certain date. (Usually July 1st in the northern emisphere and January 1st in the southern hemisphere.)

This indicator will check for missing values according to the method "from_context". Based on indice *growing_season_length()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]

- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 5.0 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold to mark the beginning and end of growing season. Default : 6.
- **mid_date** (*date (string, MM-DD)*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'. Default : 07-01.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

growing_season_length (*DataArray*) – ETCCDI Growing Season Length (Tmean > {thresh}) (growing_season_length) [days] description: {freq} number of days between the first occurrence of at least {window} consecutive days with mean daily temperature over {thresh} and the first occurrence of at least {window} consecutive days with mean daily temperature below {thresh} after {mid_date}.

Notes

Let TG_{ij} be the mean temperature at day i of period j. Then counted is the number of days between the first occurrence of at least 6 consecutive days with:

$$TG_{ij} > 5$$

and the first occurrence after 1 July of at least 6 consecutive days with:

$$TG_{ij} < 5$$

`xclim.indicators.atmos.growing_season_start(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '5.0 degC', window: int = 5, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Start of the growing season. (realm: atmos)

Day of the year of the start of a sequence of days with mean temperatures consistently above or equal to a threshold, after a period with mean temperatures consistently above the same threshold.

This indicator will check for missing values according to the method "from_context". Based on indice `growing_season_start()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 5.0 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold needed for evaluation. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

growing_season_start (*DataArray*) – Day of year of growing season start (day_of_year) description: Day of year of start of growing season, defined as the first day of consistent superior or equal to threshold temperature of {thresh} after a run of {window} days inferior to threshold temperature.

Notes

Let x_i be the daily mean temperature at day of the year i for values of i going from 1 to 365 or 366. The start date of the start of growing season is given by the smallest index i for which:

$$\prod_{j=i}^{i+w} [x_j >= thresh]$$

is true, where w is the number of days the temperature threshold should be met or exceeded, and $[P]$ is 1 if P is true, and 0 if false.

`xclim.indicators.atmos.heat_index`(*tasmax: Union[DataArray, str] = 'tasmax'*, *hurs: Union[DataArray, str] = 'hurs'*, ***, *ds: Dataset = None*) → DataArray

Daily heat index. (realm: atmos)

Perceived temperature after relative humidity is taken into account ([Blazejczyk2012]). The index is only valid for temperatures above 20°C.

Based on indice *heat_index()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **hurs** (*str or DataArray*) – Relative humidity. Default : *ds.hurs*. [Required units : []]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

heat_index (*DataArray*) – heat index (air_temperature) [C] description: Perceived temperature after relative humidity is taken into account.

Notes

While both the humidex and the heat index are calculated using dew point, the humidex uses a dew point of 7 °C (45 °F) as a base, whereas the heat index uses a dew point base of 14 °C (57 °F). Further, the heat index uses heat balance equations which account for many variables other than vapor pressure, which is used exclusively in the humidex calculation.

References

`xclim.indicators.atmos.heat_wave_frequency`(*tasmin: Union[DataArray, str] = 'tasmin'*, *tasmax: Union[DataArray, str] = 'tasmax'*, ***, *thresh_tasmin: str = '22.0 degC'*, *thresh_tasmax: str = '30 degC'*, *window: int = 3*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Heat wave frequency. (realm: atmos)

Number of heat waves over a given period. A heat wave is defined as an event where the minimum and maximum daily temperature both exceeds specific thresholds over a minimum number of days.

This indicator will check for missing values according to the method "from_context". Based on indice *heat_wave_frequency()*. Keywords : health,.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The minimum temperature threshold needed to trigger a heatwave event. Default : 22.0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The maximum temperature threshold needed to trigger a heatwave event. Default : 30 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

heat_wave_frequency (*DataArray*) – Number of heat wave events (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax} for >= {window} days) (heat_wave_events) description: {freq} number of heat wave events over a given period. An event occurs when the minimum and maximum daily temperature both exceeds specific thresholds : (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax}) over a minimum number of days ({window}).

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities ([casati2013]).

In Robinson (2001; [robinson2001]), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

`xclim.indicators.atmos.heat_wave_index(`*tasmax: Union[DataArray, str] = 'tasmax', *, thresh: str = '25.0 degC', window: int = 5, freq: str = 'YS', ds: Dataset = None*) → DataArray

Heat wave index. (realm: atmos)

Number of days that are part of a heatwave, defined as five or more consecutive days over 25℃.

This indicator will check for missing values according to the method "from_context". Based on indice *heat_wave_index()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]

- **thresh** (*quantity (string with units)*) – Threshold temperature on which to designate a heatwave. Default : 25.0 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold to qualify as a heatwave. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

heat_wave_index (*DataArray*) – Number of days that are part of a heatwave (heat_wave_index) [days] description: {freq} number of days that are part of a heatwave, defined as five or more consecutive days over {thresh}.

`xclim.indicators.atmos.heat_wave_max_length(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '22.0 degC', thresh_tasmax: str = '30 degC', window: int = 3, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Heat wave max length. (realm: atmos)

Maximum length of heat waves over a given period. A heat wave is defined as an event where the minimum and maximum daily temperature both exceeds specific thresholds over a minimum number of days.

This indicator will check for missing values according to the method "from_context". Based on indice *heat_wave_max_length()*. Keywords : health,.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The minimum temperature threshold needed to trigger a heatwave event. Default : 22.0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The maximum temperature threshold needed to trigger a heatwave event. Default : 30 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

heat_wave_max_length (*DataArray*) – Maximum length of heat wave events (Tmin > {thresh_tasmin}and Tmax > {thresh_tasmax} for >= {window} days) (spell_length_of_days_with_air_temperature_above_threshold) [days] description: {freq} maximum length of heat wave events occurring in a given period. An event occurs when the minimum and maximum daily temperature both exceeds specific thresholds (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax}) over a minimum number of days ({window}).

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities ([casati2013]).

In Robinson (2001; [robinson2001]), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

`xclim.indicators.atmos.heat_wave_total_length(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '22.0 degC', thresh_tasmax: str = '30 degC', window: int = 3, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Heat wave total length. (realm: atmos)

Total length of heat waves over a given period. A heat wave is defined as an event where the minimum and maximum daily temperature both exceeds specific thresholds over a minimum number of days. This the sum of all days in such events.

This indicator will check for missing values according to the method "from_context". Based on indice `heat_wave_total_length()`. Keywords : health,.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The minimum temperature threshold needed to trigger a heatwave event. Default : 22.0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The maximum temperature threshold needed to trigger a heatwave event. Default : 30 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

heat_wave_total_length (*DataArray*) – Total length of heat wave events (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax} for >= {window} days) (spell_length_of_days_with_air_temperature_above_threshold) [days] description: {freq} total length of heat wave events occurring in a given period. An event occurs when the minimum and maximum daily temperature both exceeds specific thresholds (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax}) over a minimum number of days ({window}).

Notes

See notes and references of *heat_wave_max_length*

`xclim.indicators.atmos.heating_degree_days`(*tas: Union[DataArray, str] = 'tas', *, thresh: str = '17.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Heating degree days. (realm: atmos)

Sum of degree days below the temperature threshold at which spaces are heated.

This indicator will check for missing values according to the method "from_context". Based on indice *heating_degree_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 17.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

heating_degree_days (*DataArray*) – Heating degree days (Tmean < {thresh}) (integral_of_air_temperature_deficit_wrt_time) [K days] cell_methods: time: sum over days description: {freq} heating degree days below {thresh}.

Notes

This index intentionally differs from its ECA&D equivalent: HD17. In HD17, values below zero are not clipped before the sum. The present definition should provide a better representation of the energy demand for heating buildings to the given threshold.

Let TG_{ij} be the daily mean temperature at day i of period j. Then the heating degree days are:

$$HD17_j = \sum_{i=1}^{I} (17 - TG_{ij}) | TG_{ij} < 17)$$

`xclim.indicators.atmos.high_precip_low_temp`(*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, pr_thresh: str = '0.4 mm/d', tas_thresh: str = '-0.2 degC', freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Number of days with precipitation above threshold and temperature below threshold. (realm: atmos)

Number of days where precipitation is greater or equal to some threshold, and temperatures are colder than some threshold. This can be used for example to identify days with the potential for freezing rain or icing conditions.

This indicator will check for missing values according to the method "from_context". Based on indice *high_precip_low_temp()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Daily mean, minimum or maximum temperature. Default : *ds.tas*. [Required units : [temperature]]
- **pr_thresh** (*quantity (string with units)*) – Precipitation threshold to exceed. Default : 0.4 mm/d. [Required units : [precipitation]]
- **tas_thresh** (*quantity (string with units)*) – Temperature threshold not to exceed. Default : -0.2 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

high_precip_low_temp (*DataArray*) – Count of days with high precipitation and low temperatures. [days] cell_methods: time: sum over days description: {freq} number of days with precipitation above {pr_thresh} and temperature below {tas_thresh}.

`xclim.indicators.atmos.hot_spell_frequency(`*tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmax: str = '30 degC', window: int = 3, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Hot spell frequency. (realm: atmos)

Number of hot spells over a given period. A hot spell is defined as an event where the maximum daily temperature exceeds a specific threshold over a minimum number of days.

This indicator will check for missing values according to the method "from_context". Based on indice *hot_spell_frequency()*. Keywords : health,.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The maximum temperature threshold needed to trigger a heatwave event. Default : 30 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

hot_spell_frequency (*DataArray*) – Number of hot spell events (Tmax > {thresh_tasmax} for >= {window} days) (hot_spell_events) description: {freq} number of hot spell events over a given period. An event occurs when the maximum daily temperature exceeds a specific threshold: (Tmax > {thresh_tasmax}) over a minimum number of days ({window}).

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities (Casati et al., 2013).

In Robinson (2001), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

Casati, B., A. Yagouti, and D. Chaumont, 2013: Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. J. Appl. Meteor. Climatol., 52, 2669–2698, https://doi.org/10.1175/JAMC-D-12-0341.1

Robinson, P.J., 2001: On the Definition of a Heat Wave. J. Appl. Meteor., 40, 762–775, https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

`xclim.indicators.atmos.hot_spell_max_length`(*tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmax: str = '30 degC', window: int = 1, freq: str = 'YS', ds: Dataset = None*) → DataArray

Longest hot spell. (realm: atmos)

Longest spell of high temperatures over a given period.

This indicator will check for missing values according to the method "from_context". Based on indice *hot_spell_max_length()*. Keywords : health,.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The maximum temperature threshold needed to trigger a heatwave event. Default : 30 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

hot_spell_max_length (*DataArray*) – Maximum length of hot spell events (Tmax > {thresh_tasmax} for >= {window} days) (spell_length_of_days_with_air_temperature_above_threshold) [days] description: {freq} maximum length of hot spell events occurring in a given period. An event occurs when the maximum daily temperature exceeds a specific threshold: (Tmax > {thresh_tasmax}) over a minimum number of days ({window}).

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities (Casati et al., 2013).

In Robinson (2001), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

Casati, B., A. Yagouti, and D. Chaumont, 2013: Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. J. Appl. Meteor. Climatol., 52, 2669–2698, https://doi.org/10.1175/JAMC-D-12-0341.1

Robinson, P.J., 2001: On the Definition of a Heat Wave. J. Appl. Meteor., 40, 762–775, https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

`xclim.indicators.atmos.huglin_index(`*tas: Union[DataArray, str] = 'tas', tasmax: Union[DataArray, str] = 'tasmax', lat: Union[DataArray, str] = 'lat', *, thresh: str = '10 degC', start_date: DayOfYearStr = '04-01', end_date: DayOfYearStr = '10-01', freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Huglin Heliothermal Index. (realm: atmos)

Growing-degree days with a base of 10°C and adjusted for latitudes between 40°N and 50°N for April to September (Northern Hemisphere; October to March in Southern Hemisphere). Originally proposed in [Huglin1978]. Used as a heat-summation metric in viticulture agroclimatology.

This indicator will check for missing values according to the method "from_context". Based on indice `huglin_index()`. With injected parameters: method=jones.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **lat** (*str or DataArray*) – Latitude coordinate. Default : *ds.lat*. [Required units : []]
- **thresh** (*quantity (string with units)*) – The temperature threshold. Default : 10 degC. [Required units : [temperature]]
- **start_date** (*date (string, MM-DD)*) – The hemisphere-based start date to consider (north = April, south = October). Default : 04-01.
- **end_date** (*date (string, MM-DD)*) – The hemisphere-based start date to consider (north = October, south = April). This date is non-inclusive. Default : 10-01.
- **freq** (*offset alias (string)*) – Resampling frequency (default: "YS"; For Southern Hemisphere, should be "AS-JUL"). Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

hi (*DataArray*) – Huglin heliothermal index (Summation of ((Tmin + Tmax)/2 - {thresh}) * Latitude-based day-lengthcoefficient (*k*), for days between {start_date} and {end_date}). description: Heat-summation index for agroclimatic suitability estimation, developed specifically for viticulture. Considers daily Tmin and Tmax with a base of {thresh}, typically between 1 April and 30 September. Integrates a day-length coefficient calculation for higher latitudes. comment: Metric originally published in Huglin (1978). Day-length coefficient based on Hall & Jones (2010)

Notes

Let TX_i and TG_i be the daily maximum and mean temperature at day i and T_{thresh} the base threshold needed for heat summation (typically, 10 degC). A day-length multiplication, k, based on latitude, lat, is also considered. Then the Huglin heliothermal index for dates between 1 April and 30 September is:

$$HI = \sum_{i=\text{April 1}}^{\text{September 30}} \left(\frac{TX_i + TG_i)}{2} - T_{thresh} \right) * k$$

For the *smoothed* method, the day-length multiplication factor, k, is calculated as follows:

$$k = f(lat) = \begin{cases} 1, & \text{if } |lat| <= 40 \\ 1 + ((abs(lat) - 40)/10) * 0.06, & \text{if } 40 < |lat| <= 50 \\ NaN, & \text{if } |lat| > 50 \end{cases}$$

For compatibility with ICCLIM, *end_ date* should be set to *11-01*, *method* should be set to *icclim*. The day-length multiplication factor, k, is calculated as follows:

$$k = f(lat) = \begin{cases} 1.0, & \text{if } |lat| <= 40 \\ 1.02, & \text{if } 40 < |lat| <= 42 \\ 1.03, & \text{if } 42 < |lat| <= 44 \\ 1.04, & \text{if } 44 < |lat| <= 46 \\ 1.05, & \text{if } 46 < |lat| <= 48 \\ 1.06, & \text{if } 48 < |lat| <= 50 \\ NaN, & \text{if } |lat| > 50 \end{cases}$$

A more robust day-length calculation based on latitude, calendar, day-of-year, and obliquity is available with *method="jones"*. See: `xclim.indices.generic.day_lengths()` or [Hall&Jones2010]_ for more information.

References

`xclim.indicators.atmos.humidex(`*tas: Union[DataArray, str] = 'tas', tdps: Optional[Union[DataArray, str]] = None, hurs: Optional[Union[DataArray, str]] = None, *, ds: Dataset = None*`)` → DataArray

Humidex index. (realm: atmos)

The humidex indicates how hot the air feels to an average person, accounting for the effect of humidity. It can be loosely interpreted as the equivalent perceived temperature when the air is dry.

Based on indice *humidex()*.

Parameters

- **tas** (*str or DataArray*) – Air temperature. Default : *ds.tas*. [Required units : [temperature]]

- **tdps** (*str or DataArray, optional*) – Dewpoint temperature. [Required units : [temperature]]
- **hurs** (*str or DataArray, optional*) – Relative humidity. [Required units : []]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
humidex (*DataArray*) – humidex index (air_temperature) [C] description: Humidex index describing the temperature felt by the average person in response to relative humidity.

Notes

The humidex is usually computed using hourly observations of dry bulb and dewpoint temperatures. It is computed using the formula based on [masterton79]:

$$T + \frac{5}{9}[e - 10]$$

where T is the dry bulb air temperature (°C). The term e can be computed from the dewpoint temperature $T_{dewpoint}$ in °K:

$$e = 6.112 \times \exp(5417.7530\left(\frac{1}{273.16} - \frac{1}{T_{\text{dewpoint}}}\right)$$

where the constant 5417.753 reflects the molecular weight of water, latent heat of vaporization, and the universal gas constant ([mekis15]). Alternatively, the term e can also be computed from the relative humidity h expressed in percent using [sirangelo20]:

$$e = \frac{h}{100} \times 6.112 * 10^{7.5T/(T+237.7)}.$$

The humidex *comfort scale* ([eccc]) can be interpreted as follows:

- 20 to 29 : no discomfort;
- 30 to 39 : some discomfort;
- 40 to 45 : great discomfort, avoid exertion;
- 46 and over : dangerous, possible heat stroke;

Please note that while both the humidex and the heat index are calculated using dew point, the humidex uses a dew point of 7 °C (45 °F) as a base, whereas the heat index uses a dew point base of 14 °C (57 °F). Further, the heat index uses heat balance equations which account for many variables other than vapor pressure, which is used exclusively in the humidex calculation.

References

`xclim.indicators.atmos.ice_days(`*tasmax: Union[DataArray, str] = 'tasmax', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of ice/freezing days. (realm: atmos)

Number of days where daily maximum temperatures are below a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *`ice_days()`*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Freezing temperature. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

ice_days (*DataArray*) – Number of ice days (Tmax < {thresh}) (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with maximum daily temperature below {thresh}.

Notes

Let TX_{ij} be the daily maximum temperature at day i of period j, and :math`TT` the threshold. Then counted is the number of days where:

$$TX_{ij} < TT$$

`xclim.indicators.atmos.jetstream_metric_woollings(`*ua: Union[DataArray, str] = 'ua', *, ds: Dataset = None*`)` → Tuple[DataArray, DataArray]

Strength and latitude of jetstream. (realm: atmos)

Identify latitude and strength of maximum smoothed zonal wind speed in the region from 15 to 75°N and -60 to 0°E, using the formula outlined in ([Woollings2010]).

Based on indice *`jetstream_metric_woollings()`*.

Parameters

- **ua** (*str or DataArray*) – Eastward wind component (u) at between 750 and 950 hPa. Default : *ds.ua*. [Required units : [speed]]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

- **jetlat** (*DataArray*) – Latitude of maximum smoothed zonal wind speed [degrees_North] description: Daily latitude of maximum smoothed zonal wind speed
- **jetstr** (*DataArray*) – Maximum strength of smoothed zonal wind speed [m s-1] description: Daily maximum strength of smoothed zonal wind speed

References

`xclim.indicators.atmos.last_snowfall`(*prsn: Union[DataArray, str] = 'prsn', *, thresh: str = '0.5 mm/day', freq: str = 'AS-JUL', ds: Dataset = None, **indexer*) → DataArray

Last day with solid precipitation above a threshold. (realm: atmos)

Returns the last day of a period where the solid precipitation exceeds a threshold. WARNING: The default freq is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *last_snowfall()*.

Parameters

- **prsn** (*str or DataArray*) – Solid precipitation flux. Default : *ds.prsn*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold precipitation flux on which to base evaluation. Default : 0.5 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

last_snowfall (*DataArray*) – Date of last snowfall (day_of_year) description: {freq} last day where the solid precipitation flux exceeded {thresh}

References

Climate Projections for the National Capital Region (2020), Volume 1: Results and Interpretation for Key Climate Indices, Report 193600.00, Prepared for Ottawa by CBCL.

`xclim.indicators.atmos.last_spring_frost`(*tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', before_date: DayOfYearStr = '07-01', window: int = 1, freq: str = 'YS', ds: Dataset = None*) → DataArray

Last day of temperatures inferior to a threshold temperature. (realm: atmos)

Returns last day of period where a temperature is inferior to a threshold over a given number of days and limited to a final calendar date.

This indicator will check for missing values according to the method "from_context". Based on indice *last_spring_frost()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **before_date** (*date (string, MM-DD)*) – Date of the year before which to look for the final frost event. Should have the format '%m-%d'. Default : 07-01.

- **window** (*number*) – Minimum number of days with temperature below threshold needed for evaluation. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

last_spring_frost (*DataArray*) – Day of year of last spring frost (day_of_year) description: Day of year of last spring frost, defined as the last day a minimum temperature threshold of {thresh} is not exceeded before a given date.

`xclim.indicators.atmos.latitude_temperature_index`(*tas: Union[DataArray, str] = 'tas', lat: Union[DataArray, str] = 'lat', *, freq: str = 'YS', ds: Dataset = None*) → DataArray

Latitude-Temperature Index. (realm: atmos)

Mean temperature of the warmest month with a latitude-based scaling factor ([Jackson&Cherry1988]_). Used for categorizing wine-growing regions.

This indicator will check for missing values according to the method "from_context". Based on indice *latitude_temperature_index()*. With injected parameters: lat_factor=60.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **lat** (*str or DataArray*) – Latitude coordinate. Default : *ds.lat*. [Required units : []]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

lti (*DataArray*) – Latitude-temperature index description: A climate indice based on mean temperature of the warmest month and a latitude-based coefficient to account for longer day-length favouring growing conditions. Developed specifically for viticulture. Mean temperature of warmest month * ({lat_factor} - latitude). comment: Indice originally published in Jackson, D. I., & Cherry, N. J. (1988)

Notes

The latitude factor of *75* is provided for examining the poleward expansion of wine-growing climates under scenarios of climate change (modified from [Kenny&Shao1992]_). For comparing 20th century/observed historical records, the original scale factor of *60* is more appropriate.

Let Tn_j be the average temperature for a given month j, lat_f be the latitude factor, and lat be the latitude of the area of interest. Then the Latitude-Temperature Index (LTI) is:

$$LTI = max(TN_j : j = 1..12)(lat_f - |lat|)$$

References

`xclim.indicators.atmos.liquid_precip_accumulation(`*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Accumulated liquid precipitation. (realm: atmos)

Resample the original daily mean precipitation flux and accumulate over each period. If a daily temperature is provided, the *phase* keyword can be used to sum precipitation of a given phase only. When the temperature is under the provided threshold, precipitation is assumed to be snow, and liquid rain otherwise. This indice is agnostic to the type of daily temperature (tas, tasmax or tasmin) given.

This indicator will check for missing values according to the method "from_context". Based on indice *precip_accumulation()*. With injected parameters: phase=liquid.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean, maximum or minimum daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold of *tas* over which the precipication is assumed to be liquid rain. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

liquidprcptot (*DataArray*) – Total liquid precipitation (lwe_thickness_of_liquid_precipitation_amount) [mm] cell_methods: time: sum over days description: {freq} total {phase} precipitation, estimated as precipitation when temperature >= {thresh}

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$

If tas and phase are given, the corresponding phase precipitation is estimated before computing the accumulation, using one of *snowfall_approximation* or *rain_approximation* with the *binary* method.

`xclim.indicators.atmos.liquid_precip_ratio(`*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', freq: str = 'QS-DEC', ds: Dataset = None, **indexer*`)` → DataArray

Ratio of rainfall to total precipitation. (realm: atmos)

The ratio of total liquid precipitation over the total precipitation. Liquid precipitation is approximated from total precipitation on days where temperature is above a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *liquid_precip_ratio()*. With injected parameters: prsn=None.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature under which precipitation is assumed to be solid. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : QS-DEC.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

liquid_precip_ratio (*DataArray*) – Ratio of rainfall to total precipitation. description: {freq} ratio of rainfall to total precipitation. Rainfall is estimated as precipitation on days where temperature is above {thresh}.

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$

$$PRwet_{ij}$$

`xclim.indicators.atmos.max_1day_precipitation_amount`(*pr: Union[DataArray, str] = 'pr', *, freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Highest 1-day precipitation amount for a period (frequency). (realm: atmos)

Resample the original daily total precipitation temperature series by taking the max over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *max_1day_precipitation_amount()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation values. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

rx1day (*DataArray*) – maximum 1-day total precipitation (lwe_thickness_of_precipitation_amount) [mm/day] cell_methods: time: maximum over days description: {freq} maximum 1-day total precipitation

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j:

$$PRx_{ij} = max(PR_{ij})$$

`xclim.indicators.atmos.max_daily_temperature_range`(*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Maximum of daily temperature range. (realm: atmos)

The mean difference between the daily maximum temperature and the daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *daily_temperature_range()*. With injected parameters: op=max.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

dtrmax (*DataArray*) – Maximum Diurnal Temperature Range (air_temperature) [K] cell_methods: time range within days time: max over days description: {freq} maximum diurnal temperature range.

Notes

For a default calculation using *op='mean'* :

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the mean diurnal temperature range in period j is:

$$DTR_j = \frac{\sum_{i=1}^{I}(TX_{ij} - TN_{ij})}{I}$$

`xclim.indicators.atmos.max_n_day_precipitation_amount`(*pr: Union[DataArray, str] = 'pr', *, window: int = 1, freq: str = 'YS', ds: Dataset = None*) → DataArray

Highest precipitation amount cumulated over a n-day moving window. (realm: atmos)

Calculate the n-day rolling sum of the original daily total precipitation series and determine the maximum value over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *max_n_day_precipitation_amount()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation values. Default : *ds.pr*. [Required units : [precipitation]]
- **window** (*number*) – Window size in days. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

rx{window}day (*DataArray*) – maximum {window}-day total precipitation (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: maximum over days description: {freq} maximum {window}-day total precipitation.

`xclim.indicators.atmos.max_pr_intensity(`*pr: Union[DataArray, str] = 'pr', *, window: int = 1, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Highest precipitation intensity over a n-hour moving window. (realm: atmos)

Calculate the n-hour rolling average of the original hourly total precipitation series and determine the maximum value over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *max_pr_intensity()*. Keywords : IDF curves.

Parameters

- **pr** (*str or DataArray*) – Hourly precipitation values. Default : *ds.pr*. [Required units : [precipitation]]
- **window** (*number*) – Window size in hours. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

max_pr_intensity (*DataArray*) – Maximum precipitation intensity over {window}h duration (precipitation) [mm/h] cell_methods: time: max description: {freq} maximum precipitation intensity over rolling {window}h window.

`xclim.indicators.atmos.maximum_consecutive_dry_days(`*pr: Union[DataArray, str] = 'pr', *, thresh: str = '1 mm/day', freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Maximum number of consecutive dry days. (realm: atmos)

Return the maximum number of consecutive days within the period where precipitation is below a certain threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *maximum_consecutive_dry_days()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold precipitation on which to base evaluation. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

cdd (*DataArray*) – Maximum consecutive dry days (Precip < {thresh}) (number_of_days_with_lwe_thickness_of_precipitation_amount_below_threshold) [days] cell_methods: time: sum over days description: {freq} maximum number of consecutive days with daily precipitation below {thresh}.

Notes

Let $\mathbf{p} = p_0, p_1, \ldots, p_n$ be a daily precipitation series and *thresh* the threshold under which a day is considered dry. Then let $\mathbf{s}$ be the sorted vector of indices i where $[p_i < thresh] \neq [p_{i+1} < thresh]$, that is, the days when the precipitation crosses the threshold. Then the maximum number of consecutive dry days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[p_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indicators.atmos.maximum_consecutive_frost_free_days`(*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None*) → DataArray

Maximum number of consecutive frost free days (Tn >= 0°C). (realm: atmos)

Return the maximum number of consecutive days within the period where the minimum temperature is above or equal to a certain threshold.

This indicator will check for missing values according to the method "from_context". Based on indice `maximum_consecutive_frost_free_days()`.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

consecutive_frost_free_days (*DataArray*) – Maximum number of consecutive days with Tmin >= {thresh} (spell_length_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: maximum over days description: {freq} maximum number of consecutive days with minimum daily temperature above or equal to {thresh}.

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily minimum temperature series and *thresh* the threshold above or equal to which a day is considered a frost free day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i <= thresh] \neq [t_{i+1} <= thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive frost free days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} >= thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indicators.atmos.maximum_consecutive_warm_days(`*tasmax: Union[DataArray, str] = 'tasmax', *, thresh: str = '25 degC', freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Maximum number of consecutive days with tasmax above a threshold (summer days). (realm: atmos)

Return the maximum number of consecutive days within the period where the maximum temperature is above a certain threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *maximum_consecutive_tx_days()*.

Parameters

- **tasmax** (*str or DataArray*) – Max daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature. Default : 25 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

maximum_consecutive_warm_days (*DataArray*) – The maximum number of days with tasmax > thresh per periods (summer days). (spell_length_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: maximum over days description: {freq} longest spell of consecutive days with Tmax above {thresh}.

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily maximum temperature series and *thresh* the threshold above which a day is considered a summer day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i < thresh] \neq [t_{i+1} < thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive dry days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indicators.atmos.maximum_consecutive_wet_days(`*pr: Union[DataArray, str] = 'pr', *, thresh: str = '1 mm/day', freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Consecutive wet days. (realm: atmos)

Returns the maximum number of consecutive wet days.

This indicator will check for missing values according to the method "from_context". Based on indice *maximum_consecutive_wet_days()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold precipitation on which to base evaluation. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

cwd (*DataArray*) – Maximum consecutive wet days (Precip >= {thresh}) (number_of_days_with_lwe_thickness_of_precipitation_amount_at_or_above_threshold) [days] cell_methods: time: sum over days description: {freq} maximum number of consecutive days with daily precipitation over {thresh}.

Notes

Let $\mathbf{x} = x_0, x_1, \ldots, x_n$ be a daily precipitation series and $\mathbf{s}$ be the sorted vector of indices i where $[p_i > thresh] \neq [p_{i+1} > thresh]$, that is, the days when the precipitation crosses the *wet day* threshold. Then the maximum number of consecutive wet days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[x_{s_j} > 0^\circ C]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

```
xclim.indicators.atmos.mean_radiant_temperature(rsds: Union[DataArray, str] = 'rsds', rsus:
                                                Union[DataArray, str] = 'rsus', rlds:
                                                Union[DataArray, str] = 'rlds', rlus:
                                                Union[DataArray, str] = 'rlus', *, stat: str =
                                                'average', ds: Dataset = None) → DataArray
```

Mean radiant temperature. (realm: atmos)

The mean radiant temperature is the incidence of radiation on the body from all directions. WARNING: There are some issues in the calculation of mrt in polar regions.

Based on indice *mean_radiant_temperature()*.

Parameters

- **rsds** (*str or DataArray*) – Surface Downwelling Shortwave Radiation Default : *ds.rsds*. [Required units : [radiation]]
- **rsus** (*str or DataArray*) – Surface Upwelling Shortwave Radiation Default : *ds.rsus*. [Required units : [radiation]]
- **rlds** (*str or DataArray*) – Surface Downwelling Longwave Radiation Default : *ds.rlds*. [Required units : [radiation]]

- **rlus** (*str or DataArray*) – Surface Upwelling Longwave Radiation Default : *ds.rlus*. [Required units : [radiation]]
- **stat** (*{'average', 'sunlit', 'instant'}*) – Which statistic to apply. If "average", the average of the cosine of the solar zenith angle is calculated. If "instant", the instantaneous cosine of the solar zenith angle is calculated. If "sunlit", the cosine of the solar zenith angle is calculated during the sunlit period of each interval. If "instant", the instantaneous cosine of the solar zenith angle is calculated. This is necessary if mrt is not None. Default : average.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

mrt (*DataArray*) – Mean radiant temperature [K] description: The incidence of radiation on the body from all directions.

Notes

This code was inspired by the *thermofeel* package.

References

Di Napoli, C., Hogan, R.J. & Pappenberger, F. Mean radiant temperature from global-scale numerical weather prediction models. Int J Biometeorol 64, 1233–1245 (2020). https://doi.org/10.1007/s00484-020-01900-5 Brimicombe , C., Di Napoli, C., Quintino, T., Pappenberger, F., Cornforth, R. and Cloke, H., 2021 thermofeel: a python thermal comfort indices library, https://doi.org/10.21957/mp6v-fd16

`xclim.indicators.atmos.potential_evapotranspiration`(*tasmin: Optional[Union[DataArray, str]] = None, tasmax: Optional[Union[DataArray, str]] = None, tas: Optional[Union[DataArray, str]] = None, lat: Optional[Union[DataArray, str]] = None, *, method: str = 'BR65', peta: float | None = 0.00516409319477, petb: float | None = 0.0874972822289, ds: Dataset = None*) → DataArray

Potential evapotranspiration. (realm: atmos)

The potential for water evaporation from soil and transpiration by plants if the water supply is sufficient, according to a given method.

Based on indice *`potential_evapotranspiration()`*.

Parameters

- **tasmin** (*str or DataArray, optional*) – Minimum daily temperature. [Required units : [temperature]]
- **tasmax** (*str or DataArray, optional*) – Maximum daily temperature. [Required units : [temperature]]
- **tas** (*str or DataArray, optional*) – Mean daily temperature. [Required units : [temperature]]
- **lat** (*str or DataArray, optional*) – Latitude. If not given, it is sought on tasmin or tas with cf-xarray. [Required units : []]

- **method** (*{'hargreaves85', 'baierrobertson65', 'HG85', 'MB05', 'mcguinnessbordne05', 'thornthwaite48', 'BR65', 'TW48'}*) – Which method to use, see notes. Default : BR65.
- **peta** (*number*) – Used only with method MB05 as a for calculation of PET, see Notes section. Default value resulted from calibration of PET over the UK. Default : 0.00516409319477.
- **petb** (*number*) – Used only with method MB05 as b for calculation of PET, see Notes section. Default value resulted from calibration of PET over the UK. Default : 0.0874972822289.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

evspsblpot (*DataArray*) – Potential evapotranspiration (water_potential_evapotranspiration_flux) [kg m-2 s-1] description: The potential for water evaporation from soil and transpiration by plants if the water supply is sufficient, with the method {method}.

Notes

Available methods are:

- "baierrobertson65" or "BR65", based on [BaierRobertson1965]. Requires tasmin and tasmax, daily [D] freq.
- "hargreaves85" or "HG85", based on [Hargreaves1985]. Requires tasmin and tasmax, daily [D] freq. (optional: tas can be given in addition of tasmin and tasmax).
- "mcguinnessbordne05" or "MB05", based on [Tanguy2018]. Requires tas, daily [D] freq, with latitudes 'lat'.
- "thornthwaite48" or "TW48", based on [Thornthwaite1948]. Requires tasmin and tasmax, monthly [MS] or daily [D] freq. (optional: tas can be given instead of tasmin and tasmax).

The McGuinness-Bordne [McGuinness1972] equation is:

$$PET[mmday^{-1}] = a * \frac{S_0}{\lambda}T_a + b * S_0\lambda$$

where a and b are empirical parameters; S_0 is the extraterrestrial radiation [MJ m-2 day-1], assuming a solar constant of 1367 W m-2;
$lambda$ is the latent heat of vaporisation [MJ kg-1] and T_a is the air temperature [°C]. The equation was originally derived for the USA, with $a = 0.0147$ and $b = 0.07353$. The default parameters used here are calibrated for the UK, using the method described in [Tanguy2018].

Methods "BR65", "HG85" and "MB05" use an approximation of the extraterrestrial radiation. See `extraterrestrial_solar_radiation()`.

References

`xclim.indicators.atmos.precip_accumulation`(*pr: Union[DataArray, str] = 'pr', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Accumulated total precipitation (solid and liquid) (realm: atmos)

Resample the original daily mean precipitation flux and accumulate over each period. If a daily temperature is provided, the *phase* keyword can be used to sum precipitation of a given phase only. When the temperature is under the provided threshold, precipitation is assumed to be snow, and liquid rain otherwise. This indice is agnostic to the type of daily temperature (tas, tasmax or tasmin) given.

This indicator will check for missing values according to the method "from_context". Based on indice *precip_accumulation()*. With injected parameters: tas=None, phase=None.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold of *tas* over which the precipication is assumed to be liquid rain. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

prcptot (*DataArray*) – Total precipitation (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum over days description: {freq} total precipitation

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$

If tas and phase are given, the corresponding phase precipitation is estimated before computing the accumulation, using one of *snowfall_approximation* or *rain_approximation* with the *binary* method.

`xclim.indicators.atmos.rain_approximation`(*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', method: str = 'binary', ds: Dataset = None*) → DataArray

Rainfall approximation from total precipitation and temperature. (realm: atmos)

Liquid precipitation estimated from precipitation and temperature according to a given method. This is a convenience method based on `snowfall_approximation()`, see the latter for details.

Based on indice *rain_approximation()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]

- **tas** (*str or DataArray*) – Mean, maximum, or minimum daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature, used by method "binary". Default : 0 degC. [Required units : [temperature]]
- **method** (*{'brown', 'auer', 'binary'}*) – Which method to use when approximating snowfall from total precipitation. See notes. Default : binary.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

prlp (*DataArray*) – Liquid precipitation (precipitation_flux) [kg m-2 s-1] description: Liquid precipitation estimated from total precipitation and temperature with method {method} and threshold temperature {thresh}.

Notes

This method computes the snowfall approximation and subtracts it from the total precipitation to estimate the liquid rain precipitation.

`xclim.indicators.atmos.rain_on_frozen_ground_days(`*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, thresh: str = '1 mm/d', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of rain on frozen ground events. (realm: atmos)

Number of days with rain above a threshold after a series of seven days below freezing temperature. Precipitation is assumed to be rain when the temperature is above 0℃.

This indicator will check for missing values according to the method "from_context". Based on indice *rain_on_frozen_ground_days()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Precipitation threshold to consider a day as a rain event. Default : 1 mm/d. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

rain_frzgr (*DataArray*) – Number of rain on frozen ground days (number_of_days_with_lwe_thickness_of_precipitation_amount_above_threshold) [days] description: {freq} number of days with rain above {thresh} after a series of seven days with average daily temperature below 0℃. Precipitation is assumed to be rain when thedaily average temperature is above 0℃.

Notes

Let PR_i be the mean daily precipitation and TG_i be the mean daily temperature of day i. Then for a period j, rain on frozen grounds days are counted where:

$$PR_i > Threshold[mm]$$

and where

$$TG_i 0$$

is true for continuous periods where $i 7$

`xclim.indicators.atmos.relative_humidity(`*tas: Union[DataArray, str] = 'tas', huss: Union[DataArray, str] = 'huss', ps: Union[DataArray, str] = 'ps', *, ice_thresh: str = None, method: str = 'sonntag90', ds: Dataset = None*`)` → DataArray

Relative humidity from temperature, pressure and specific humidity. (realm: atmos)

Compute relative humidity from temperature and either dewpoint temperature or specific humidity and pressure through the saturation vapor pressure.

Based on indice `relative_humidity()`. With injected parameters: tdps=None, invalid_values=mask.

Parameters

- **tas** (*str or DataArray*) – Temperature array Default : *ds.tas*. [Required units : [temperature]]
- **huss** (*str or DataArray*) – Specific humidity. Default : *ds.huss*. [Required units : []]
- **ps** (*str or DataArray*) – Air Pressure. Default : *ds.ps*. [Required units : [pressure]]
- **ice_thresh** (*quantity (string with units)*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water. Does nothing if 'method' is "bohren98". Default : None. [Required units : [temperature]]
- **method** (*{'goffgratch46', 'wmo08', 'sonntag90', 'bohren98', 'tetens30'}*) – Which method to use, see notes of this function and of *saturation_vapor_pressure*. Default : sonntag90.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

hurs (*DataArray*) – Relative Humidity (relative_humidity) [%] description: <Dynamically generated string>

Notes

In the following, let T, T_d, q and p be the temperature, the dew point temperature, the specific humidity and the air pressure.

For the "bohren98" method : This method does not use the saturation vapor pressure directly, but rather uses an approximation of the ratio of $\frac{e_{sat}(T_d)}{e_{sat}(T)}$. With L the enthalpy of vaporization of water and R_w the gas constant for water vapor, the relative humidity is computed as:

$$RH = e^{\frac{-L(T-T_d)}{R_w T T_d}}$$

From [BohrenAlbrecht1998], formula taken from [Lawrence2005]. $L = 2.5 \times 10^{-6}$ J kg-1, exact for $T = 273.15$ K, is used.

Other methods: With w, w_{sat}, e_{sat} the mixing ratio, the saturation mixing ratio and the saturation vapor pressure. If the dewpoint temperature is given, relative humidity is computed as:

$$RH = 100\frac{e_{sat}(T_d)}{e_{sat}(T)}$$

Otherwise, the specific humidity and the air pressure must be given so relative humidity can be computed as:

$$RH = 100\frac{w}{w_{sat}}w = \frac{q}{1-q}w_{sat} = 0.622\frac{e_{sat}}{P - e_{sat}}$$

The methods differ by how e_{sat} is computed. See the doc of `xclim.core.utils.saturation_vapor_pressure()`.

References

`xclim.indicators.atmos.relative_humidity_from_dewpoint(`*tas: Union[DataArray, str] = 'tas', tdps: Union[DataArray, str] = 'tdps', *, ice_thresh: str = None, method: str = 'sonntag90', ds: Dataset = None*`)` → DataArray

Relative humidity from temperature and dewpoint temperature. (realm: atmos)

Compute relative humidity from temperature and either dewpoint temperature or specific humidity and pressure through the saturation vapor pressure.

Based on indice *relative_humidity()*. With injected parameters: huss=None, ps=None, invalid_values=mask.

Parameters

- **tas** (*str or DataArray*) – Temperature array Default : *ds.tas*. [Required units : [temperature]]
- **tdps** (*str or DataArray*) – Dewpoint temperature, if specified, overrides huss and ps. Default : *ds.tdps*. [Required units : [temperature]]
- **ice_thresh** (*quantity (string with units)*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water. Does nothing if 'method' is "bohren98". Default : None. [Required units : [temperature]]
- **method** (*{'goffgratch46', 'wmo08', 'sonntag90', 'bohren98', 'tetens30'}*) – Which method to use, see notes of this function and of *saturation_vapor_pressure*. Default : sonntag90.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

hurs (*DataArray*) – Relative Humidity (relative_humidity) [%] description: <Dynamically generated string>

Notes

In the following, let T, T_d, q and p be the temperature, the dew point temperature, the specific humidity and the air pressure.

For the "bohren98" method : This method does not use the saturation vapor pressure directly, but rather uses an approximation of the ratio of $\frac{e_{sat}(T_d)}{e_{sat}(T)}$. With L the enthalpy of vaporization of water and R_w the gas constant for water vapor, the relative humidity is computed as:

$$RH = e^{\frac{-L(T-T_d)}{R_w T T_d}}$$

From [BohrenAlbrecht1998], formula taken from [Lawrence2005]. $L = 2.5 \times 10^{-6}$ J kg-1, exact for $T = 273.15$ K, is used.

Other methods: With w, w_{sat}, e_{sat} the mixing ratio, the saturation mixing ratio and the saturation vapor pressure. If the dewpoint temperature is given, relative humidity is computed as:

$$RH = 100\frac{e_{sat}(T_d)}{e_{sat}(T)}$$

Otherwise, the specific humidity and the air pressure must be given so relative humidity can be computed as:

$$RH = 100\frac{w}{w_{sat}} w = \frac{q}{1-q} w_{sat} = 0.622\frac{e_{sat}}{P - e_{sat}}$$

The methods differ by how e_{sat} is computed. See the doc of `xclim.core.utils.saturation_vapor_pressure()`.

References

`xclim.indicators.atmos.rprctot`(*pr: Union[DataArray, str] = 'pr', prc: Union[DataArray, str] = 'prc', *, thresh: str = '1.0 mm/day', freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Proportion of accumulated precipitation arising from convective processes. (realm: atmos)

Return the proportion of total accumulated precipitation due to convection on days with total precipitation exceeding a specified threshold during the given period.

This indicator will check for missing values according to the method "from_context". Based on indice *rprctot()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **prc** (*str or DataArray*) – Daily convective precipitation. Default : *ds.prc*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1.0 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

rprctot (*DataArray*) – The proportion of the total precipitation accounted for by convective precipitation for each period. cell_methods: time: sum description: Proportion of accumulated precipitation arising from convective processes.

`xclim.indicators.atmos.saturation_vapor_pressure`(*tas: Union[DataArray, str] = 'tas'*, ***, *ice_thresh: str = None*, *method: str = 'sonntag90'*, *ds: Dataset = None*) → DataArray

Saturation vapor pressure from temperature. (realm: atmos)

Based on indice *saturation_vapor_pressure()*.

Parameters

- **tas** (*str or DataArray*) – Temperature array. Default : *ds.tas*. [Required units : [temperature]]
- **ice_thresh** (*quantity (string with units)*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water. Default : None. [Required units : [temperature]]
- **method** (*{'goffgratch46', 'wmo08', 'sonntag90', 'its90', 'tetens30'}*) – Which method to use, see notes. Default : sonntag90.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

e_sat (*DataArray*) – Saturation vapor pressure [Pa] description: <Dynamically generated string>

Notes

In all cases implemented here $log(e_{sat})$ is an empirically fitted function (usually a polynomial) where coefficients can be different when ice is taken as reference instead of water. Available methods are:

- "goffgratch46" or "GG46", based on [goffgratch46], values and equation taken from [voemel].
- "sonntag90" or "SO90", taken from [sonntag90].
- "tetens30" or "TE30", based on [tetens30], values and equation taken from [voemel].
- "wmo08" or "WMO08", taken from [wmo08].
- "its90" or "ITS90", taken from [its90].

References

`xclim.indicators.atmos.snowfall_approximation`(*pr: Union[DataArray, str] = 'pr'*, *tas: Union[DataArray, str] = 'tas'*, ***, *thresh: str = '0 degC'*, *method: str = 'binary'*, *ds: Dataset = None*) → DataArray

Snowfall approximation from total precipitation and temperature. (realm: atmos)

Solid precipitation estimated from precipitation and temperature according to a given method.

Based on indice *snowfall_approximation()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean, maximum, or minimum daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature, used by method "binary". Default : 0 degC. [Required units : [temperature]]
- **method** (*{'brown', 'auer', 'binary'}*) – Which method to use when approximating snowfall from total precipitation. See notes. Default : binary.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

prsn (*DataArray*) – Solid precipitation (solid_precipitation_flux) [kg m-2 s-1] description: Solid precipitation estimated from total precipitation and temperature with method {method} and threshold temperature {thresh}.

Notes

The following methods are available to approximate snowfall and are drawn from the Canadian Land Surface Scheme (CLASS, [Verseghy09]).

- `'binary'` : When the temperature is under the freezing threshold, precipitation is assumed to be solid. The method is agnostic to the type of temperature used (mean, maximum or minimum).
- `'brown'` : The phase between the freezing threshold goes from solid to liquid linearly over a range of 2°C over the freezing point.
- `'auer'` : The phase between the freezing threshold goes from solid to liquid as a degree six polynomial over a range of 6°C over the freezing point.

References

https://gitlab.com/cccma/classic/-/blob/master/src/atmosphericVarsCalc.f90

`xclim.indicators.atmos.solid_precip_accumulation`(*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Accumulated solid precipitation. (realm: atmos)

Resample the original daily mean precipitation flux and accumulate over each period. If a daily temperature is provided, the *phase* keyword can be used to sum precipitation of a given phase only. When the temperature is under the provided threshold, precipitation is assumed to be snow, and liquid rain otherwise. This indice is agnostic to the type of daily temperature (tas, tasmax or tasmin) given.

This indicator will check for missing values according to the method "from_context". Based on indice `precip_accumulation()`. With injected parameters: phase=solid.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean, maximum or minimum daily temperature. Default : *ds.tas*. [Required units : [temperature]]

- **thresh** (*quantity (string with units)*) – Threshold of *tas* over which the precipication is assumed to be liquid rain. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

solidprcptot (*DataArray*) – Total solid precipitation (lwe_thickness_of_snowfall_amount) [mm] cell_methods: time: sum over days description: {freq} total solid precipitation, estimated as precipitation when temperature < {thresh}

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$

If tas and phase are given, the corresponding phase precipitation is estimated before computing the accumulation, using one of *snowfall_approximation* or *rain_approximation* with the *binary* method.

`xclim.indicators.atmos.specific_humidity(`*tas: Union[DataArray, str] = 'tas', hurs: Union[DataArray, str] = 'hurs', ps: Union[DataArray, str] = 'ps', *, ice_thresh: str = None, method: str = 'sonntag90', ds: Dataset = None*`)` → DataArray

Specific humidity from temperature, relative humidity and pressure. (realm: atmos)

Specific humidity is the ratio between the mass of water vapour and the mass of moist air [WMO08].

Based on indice *specific_humidity()*. With injected parameters: invalid_values=mask.

Parameters

- **tas** (*str or DataArray*) – Temperature array Default : *ds.tas*. [Required units : [temperature]]
- **hurs** (*str or DataArray*) – Relative Humidity. Default : *ds.hurs*. [Required units : []]
- **ps** (*str or DataArray*) – Air Pressure. Default : *ds.ps*. [Required units : [pressure]]
- **ice_thresh** (*quantity (string with units)*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water. Default : None. [Required units : [temperature]]
- **method** (*{'wmo08', 'goffgratch46', 'tetens30', 'sonntag90'}*) – Which method to use, see notes of this function and of *saturation_vapor_pressure*. Default : sonntag90.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

huss (*DataArray*) – Specific Humidity (specific_humidity) description: <Dynamically generated string>

Notes

In the following, let T, $hurs$ (in %) and p be the temperature, the relative humidity and the air pressure. With w, w_{sat}, e_{sat} the mixing ratio, the saturation mixing ratio and the saturation vapor pressure, specific humidity q is computed as:

$$w_{sat} = 0.622\frac{e_{sat}}{P - e_{sat}}w = w_{sat} * hurs/100q = w/(1 + w)$$

The methods differ by how e_{sat} is computed. See the doc of *xclim.core.utils.saturation_vapor_pressure*.

If *invalid_values* is not *None*, the saturation specific humidity q_{sat} is computed as:

$$q_{sat} = w_{sat}/(1 + w_{sat})$$

References

`xclim.indicators.atmos.specific_humidity_from_dewpoint(`*tdps: Union[DataArray, str] = 'tdps', ps: Union[DataArray, str] = 'ps', *, method: str = 'sonntag90', ds: Dataset = None*`)` → DataArray

Specific humidity from dewpoint temperature and air pressure. (realm: atmos)

Specific humidity is the ratio between the mass of water vapour and the mass of moist air [WMO08].

Based on indice `specific_humidity_from_dewpoint()`.

Parameters

- **tdps** (*str or DataArray*) – Dewpoint temperature array. Default : *ds.tdps*. [Required units : [temperature]]
- **ps** (*str or DataArray*) – Air pressure array. Default : *ds.ps*. [Required units : [pressure]]
- **method** (*{'wmo08', 'goffgratch46', 'tetens30', 'sonntag90'}*) – Method to compute the saturation vapor pressure. Default : sonntag90.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

huss_fromdewpoint (*DataArray*) – Specific Humidity (specific_humidity) description: Computed from dewpoint temperature and pressure through the saturation vapor pressure, which was calculated according to the {method} method.

Notes

If e is the water vapor pressure, and p the total air pressure, then specific humidity is given by

$$q = m_w e/(m_a(p - e) + m_w e)$$

where m_w and m_a are the molecular weights of water and dry air respectively. This formula is often written with $= m_w/m_a$, which simplifies to $q = e/(p - e(1-))$.

References

`xclim.indicators.atmos.tg(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, ds: Dataset = None*`)` → DataArray

Average temperature from minimum and maximum temperatures. (realm: atmos)

We assume a symmetrical distribution for the temperature and retrieve the average value as Tg = (Tx + Tn) / 2

Based on indice *tas()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum (daily) temperature Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum (daily) temperature Default : *ds.tasmax*. [Required units : [temperature]]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

tg (*DataArray*) – Daily mean temperature (air_temperature) [K] cell_methods: time: mean within days description: Estimated mean temperature from maximum and minimum temperatures

`xclim.indicators.atmos.tg10p(`*tas: Union[DataArray, str] = 'tas', tas_per: Union[DataArray, str] = 'tas_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily mean temperature below the 10th percentile. (realm: atmos)

Number of days with daily mean temperature below the 10th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tg10p()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **tas_per** (*str or DataArray*) – 10th percentile of daily mean temperature. Default : *ds.tas_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tg10p (*DataArray*) – Number of days when Tmean < {tas_per_thresh}th percentile (days_with_air_temperature_below_threshold) [days] cell_methods: time:

sum over days description: {freq} number of days with mean daily temperature below the {tas_per_thresh}th percentile(s). A {tas_per_window} day(s) window, centred on each calendar day in the {tas_per_period} period, is used to compute the {tas_per_thresh}th percentile(s).

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

`xclim.indicators.atmos.tg90p(`*tas: Union[DataArray, str] = 'tas', tas_per: Union[DataArray, str] = 'tas_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily mean temperature over the 90th percentile. (realm: atmos)

Number of days with daily mean temperature over the 90th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tg90p()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **tas_per** (*str or DataArray*) – 90th percentile of daily mean temperature. Default : *ds.tas_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tg90p (*DataArray*) – Number of days when Tmean > {tas_per_thresh}th percentile (days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with mean daily temperature above the the {tas_per_thresh}th percentile(s). A {tas_per_window} day(s) window, centred on each calendar day in the {tas_per_period} period, is used to compute the {tas_per_thresh}th percentile(s).

Notes

The 90th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

`xclim.indicators.atmos.tg_days_above`(*tas: Union[DataArray, str] = 'tas'*, ***, *thresh: str = '10.0 degC'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of days with tas above a threshold. (realm: atmos)

Number of days where daily mean temperature exceeds a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tg_days_above()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 10.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tg_days_above (*DataArray*) – Number of days with Tavg > {thresh} (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days where daily mean temperature exceeds {thresh}.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then counted is the number of days where:

$$TG_{ij} > Threshold[]$$

`xclim.indicators.atmos.tg_days_below`(*tas: Union[DataArray, str] = 'tas'*, ***, *thresh: str = '10.0 degC'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of days with tas below a threshold. (realm: atmos)

Number of days where daily mean temperature is below a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tg_days_below()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 10.0 degC. [Required units : [temperature]]

- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
tg_days_below (*DataArray*) – Number of days with Tavg < {thresh} (number_of_days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days where daily mean temperature is below {thresh}.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then counted is the number of days where:

$$TG_{ij} < Threshold[]$$

`xclim.indicators.atmos.tg_max`(*tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Highest mean temperature. (realm: atmos)

The maximum of daily mean temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tg_max()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
tg_max (*DataArray*) – Maximum daily mean temperature (air_temperature) [K] cell_methods: time: maximum over days description: {freq} maximum of daily mean temperature.

Notes

Let TN_{ij} be the mean temperature at day i of period j. Then the maximum daily mean temperature for period j is:

$$TNx_j = max(TN_{ij})$$

`xclim.indicators.atmos.tg_mean(`*tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Mean of daily average temperature. (realm: atmos)

Resample the original daily mean temperature series by taking the mean over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *tg_mean()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tg_mean (*DataArray*) – Mean daily mean temperature (air_temperature) [K] cell_methods: time: mean over days description: {freq} mean of daily mean temperature.

Notes

Let TN_i be the mean daily temperature of day i, then for a period p starting at day a and finishing on day b:

$$TG_p = \frac{\sum_{i=a}^{b} TN_i}{b - a + 1}$$

`xclim.indicators.atmos.tg_min(`*tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Lowest mean temperature. (realm: atmos)

Minimum of daily mean temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tg_min()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tg_min (*DataArray*) – Minimum daily mean temperature (air_temperature) [K] cell_methods: time: minimum over days description: {freq} minimum of daily mean temperature.

Notes

Let TG_{ij} be the mean temperature at day i of period j. Then the minimum daily mean temperature for period j is:

$$TGn_j = min(TG_{ij})$$

`xclim.indicators.atmos.thawing_degree_days(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Growing degree-days over threshold temperature value. (realm: atmos)

The sum of degree-days over the threshold temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *growing_degree_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

thawing_degree_days (*DataArray*) – Thawing degree days (degree days above 0°C) (integral_of_air_temperature_excess_wrt_time) [K days] cell_methods: time: sum over days description: {freq} thawing degree days above 0°C.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then the growing degree days are:

$$GD4_j = \sum_{i=1}^{I}(TG_{ij} - 4 | TG_{ij} > 4)$$

`xclim.indicators.atmos.tn10p(`*tasmin: Union[DataArray, str] = 'tasmin', tasmin_per: Union[DataArray, str] = 'tasmin_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily minimum temperature below the 10th percentile. (realm: atmos)

Number of days with daily minimum temperature below the 10th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tn10p()*.

Parameters

- **tasmin** (*str or DataArray*) – Mean daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]

- **tasmin_per** (*str or DataArray*) – 10th percentile of daily minimum temperature. Default : *ds.tasmin_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn10p (*DataArray*) – Number of days when Tmin < {tasmin_per_thresh}th percentile (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with minimum daily temperature below the the {tasmin_per_thresh}th percentile(s). A {tasmin_per_window} day(s) window, centred on each calendar day in the {tasmin_per_period} period, is used to compute the {tasmin_per_thresh}th percentile(s).

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

`xclim.indicators.atmos.tn90p(`*tasmin: Union[DataArray, str] = 'tasmin', tasmin_per: Union[DataArray, str] = 'tasmin_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily minimum temperature over the 90th percentile. (realm: atmos)

Number of days with daily minimum temperature over the 90th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tn90p()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmin_per** (*str or DataArray*) – 90th percentile of daily minimum temperature. Default : *ds.tasmin_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn90p (*DataArray*) – Number of days when Tmin > {tasmin_per_thresh}th percentile (days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with minimum daily temperature above the the {tasmin_per_thresh}th percentile(s). A {tasmin_per_window} day(s) window, centred on each calendar day in the {tasmin_per_period} period, is used to compute the {tasmin_per_thresh}th percentile(s).

Notes

The 90th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

`xclim.indicators.atmos.tn_days_above(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '20.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with tasmin above a threshold (number of tropical nights). (realm: atmos)

Number of days where daily minimum temperature exceeds a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_days_above()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 20.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn_days_above (*DataArray*) – Number of days with Tmin > {thresh} (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days where daily minimum temperature exceeds {thresh}.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} > Threshold[]$$

`xclim.indicators.atmos.tn_days_below(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '-10.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with tasmin below a threshold. (realm: atmos)

Number of days where daily minimum temperature is below a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_days_below()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : -10.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn_days_below (*DataArray*) – Number of days with Tmin < {thresh} (number_of_days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days where daily minimum temperature is below {thresh}.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} < Threshold[]$$

`xclim.indicators.atmos.tn_max(`*tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Highest minimum temperature. (realm: atmos)

The maximum of daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_max()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]

- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn_max (*DataArray*) – Maximum daily minimum temperature (air_temperature) [K] cell_methods: time: maximum over days description: {freq} maximum of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then the maximum daily minimum temperature for period j is:

$$TNx_j = max(TN_{ij})$$

`xclim.indicators.atmos.tn_mean(`*tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Mean minimum temperature. (realm: atmos)

Mean of daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_mean()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn_mean (*DataArray*) – Mean daily minimum temperature (air_temperature) [K] cell_methods: time: mean over days description: {freq} mean of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then mean values in period j are given by:

$$TN_{ij} = \frac{\sum_{i=1}^{I} TN_{ij}}{I}$$

`xclim.indicators.atmos.tn_min`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Lowest minimum temperature. (realm: atmos)

Minimum of daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_min()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn_min (*DataArray*) – Minimum daily minimum temperature (air_temperature) [K] cell_methods: time: minimum over days description: {freq} minimum of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then the minimum daily minimum temperature for period j is:

$$TNn_j = min(TN_{ij})$$

`xclim.indicators.atmos.tropical_nights`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *thresh: str = '20.0 degC'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of days with tasmin above a threshold (number of tropical nights). (realm: atmos)

Number of days where daily minimum temperature exceeds a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_days_above()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 20.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tropical_nights (*DataArray*) – Number of Tropical Nights (Tmin > {thresh}) (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of Tropical Nights : defined as days with minimum daily temperature above {thresh}.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} > Threshold[]$$

`xclim.indicators.atmos.tx10p(`*tasmax: Union[DataArray, str] = 'tasmax', tasmax_per: Union[DataArray, str] = 'tasmax_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily maximum temperature below the 10th percentile. (realm: atmos)

Number of days with daily maximum temperature below the 10th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tx10p()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **tasmax_per** (*str or DataArray*) – 10th percentile of daily maximum temperature. Default : *ds.tasmax_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx10p (*DataArray*) – Number of days when Tmax < {tasmax_per_thresh}th percentile (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with maximum daily temperature below the {tasmax_per_thresh}th percentile(s). A {tasmax_per_window} day(s) window, centred on each calendar day in the {tasmax_per_period} period, is used to compute the {tasmax_per_thresh}th percentile(s).

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

`xclim.indicators.atmos.tx90p(`*tasmax: Union[DataArray, str] = 'tasmax', tasmax_per: Union[DataArray, str] = 'tasmax_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily maximum temperature over the 90th percentile. (realm: atmos)

Number of days with daily maximum temperature over the 90th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tx90p()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **tasmax_per** (*str or DataArray*) – 90th percentile of daily maximum temperature. Default : *ds.tasmax_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx90p (*DataArray*) – Number of days when Tmax > {tasmax_per_thresh}th percentile (days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with maximum daily temperature above the {tasmax_per_thresh}th percentile(s). A {tasmax_per_window} day(s) window, centred on each calendar day in the {tasmax_per_period} period, is used to compute the {tasmax_per_thresh}th percentile(s).

Notes

The 90th percentile should be computed for a 5-day window centered on each calendar day for a reference period.

`xclim.indicators.atmos.tx_days_above(`*tasmax: Union[DataArray, str] = 'tasmax', *, thresh: str = '25.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with tasmax above a threshold (number of summer days). (realm: atmos)

Number of days where daily maximum temperature exceeds a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_days_above()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 25.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx_days_above (*DataArray*) – Number of days with Tmax > {thresh} (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days where daily maximum temperature exceeds {thresh}.

Notes

Let TX_{ij} be the daily maximum temperature at day i of period j. Then counted is the number of days where:

$$TX_{ij} > Threshold[]$$

`xclim.indicators.atmos.tx_days_below(`*tasmax: Union[DataArray, str] = 'tasmax', *, thresh: str = '25.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with tmax below a threshold. (realm: atmos)

Number of days where daily maximum temperature is below a threshold.

This indicator will check for missing values according to the method “from_context”. Based on indice *tx_days_below()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 25.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx_days_below (*DataArray*) – Number of days with Tmax < {thresh} (number_of_days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days where daily max temperature is below {thresh}.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} < Threshold[]$$

`xclim.indicators.atmos.tx_max(`*tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Highest max temperature. (realm: atmos)

The maximum value of daily maximum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_max()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx_max (*DataArray*) – Maximum daily maximum temperature (air_temperature) [K] cell_methods: time: maximum over days description: {freq} maximum of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then the maximum daily maximum temperature for period j is:

$$TXx_j = max(TX_{ij})$$

`xclim.indicators.atmos.tx_mean(`*tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Mean max temperature. (realm: atmos)

The mean of daily maximum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_mean()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
tx_mean (*DataArray*) – Mean daily maximum temperature (air_temperature) [K] cell_methods: time: mean over days description: {freq} mean of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then mean values in period j are given by:

$$TX_{ij} = \frac{\sum_{i=1}^{I} TX_{ij}}{I}$$

`xclim.indicators.atmos.tx_min(`*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*`)` → DataArray

Lowest max temperature. (realm: atmos)

The minimum of daily maximum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_min()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
tx_min (*DataArray*) – Minimum daily maximum temperature (air_temperature) [K] cell_methods: time: minimum over days description: {freq} minimum of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then the minimum daily maximum temperature for period j is:

$$TXn_j = min(TX_{ij})$$

`xclim.indicators.atmos.tx_tn_days_above(`*tasmin: Union[DataArray, str] = 'tasmin'*, *tasmax: Union[DataArray, str] = 'tasmax'*, ***, *thresh_tasmin: str = '22 degC'*, *thresh_tasmax: str = '30 degC'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*`)` → DataArray

Number of days with both hot maximum and minimum daily temperatures. (realm: atmos)

The number of days per period with tasmin above a threshold and tasmax above another threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_tn_days_above()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – Threshold temperature for tasmin on which to base evaluation. Default : 22 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – Threshold temperature for tasmax on which to base evaluation. Default : 30 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx_tn_days_above (*DataArray*) – Number of days with Tmax > {thresh_tasmax} and Tmin > {thresh_tasmin} (number_of_days_with_air_temperature_above_threshold) [days] description: {freq} number of days where daily maximum temperature exceeds {thresh_tasmax} and minimum temperature exceeds {thresh_tasmin}.

Notes

Let TX_{ij} be the maximum temperature at day i of period j, TN_{ij} the daily minimum temperature at day i of period j, TX_{thresh} the threshold for maximum daily temperature, and TN_{thresh} the threshold for minimum daily temperature. Then counted is the number of days where:

$$TX_{ij} > TX_{thresh}[]$$

and where:

$$TN_{ij} > TN_{thresh}[]$$

`xclim.indicators.atmos.universal_thermal_climate_index(`*tas: Union[DataArray, str] = 'tas', hurs: Union[DataArray, str] = 'hurs', sfcWind: Union[DataArray, str] = 'sfcWind', mrt: Optional[Union[DataArray, str]] = None, rsds: Optional[Union[DataArray, str]] = None, rsus: Optional[Union[DataArray, str]] = None, rlds: Optional[Union[DataArray, str]] = None, rlus: Optional[Union[DataArray, str]] = None, *, stat: str = 'average', mask_invalid: bool = True, ds: Dataset = None*) → DataArray

Universal thermal climate index. (realm: atmos)

The UTCI is the equivalent temperature for the environment derived from a reference environment and is used to evaluate heat stress in outdoor spaces.

Based on indice `universal_thermal_climate_index()`.

Parameters

- **tas** (*str or DataArray*) – Mean temperature Default : *ds.tas*. [Required units : [temperature]]
- **hurs** (*str or DataArray*) – Relative Humidity Default : *ds.hurs*. [Required units : []]
- **sfcWind** (*str or DataArray*) – Wind velocity Default : *ds.sfcWind*. [Required units : [speed]]
- **mrt** (*str or DataArray, optional*) – Mean radiant temperature [Required units : [temperature]]
- **rsds** (*str or DataArray, optional*) – Surface Downwelling Shortwave Radiation This is necessary if mrt is not None. [Required units : [radiation]]
- **rsus** (*str or DataArray, optional*) – Surface Upwelling Shortwave Radiation This is necessary if mrt is not None. [Required units : [radiation]]
- **rlds** (*str or DataArray, optional*) – Surface Downwelling Longwave Radiation This is necessary if mrt is not None. [Required units : [radiation]]
- **rlus** (*str or DataArray, optional*) – Surface Upwelling Longwave Radiation This is necessary if mrt is not None. [Required units : [radiation]]
- **stat** (*{'average', 'sunlit', 'instant'}*) – Which statistic to apply. If "average", the average of the cosine of the solar zenith angle is calculated. If "instant", the instantaneous cosine of the solar zenith angle is calculated. If "sunlit", the cosine of the solar zenith angle is calculated during the sunlit period of each interval. If "instant", the instantaneous cosine of the solar zenith angle is calculated. This is necessary if mrt is not None. Default : average.
- **mask_invalid** (*boolean*) – If True (default), UTCI values are NaN where any of the inputs are outside their validity ranges : -50°C < tas < 50°C, -30°C < tas - mrt < 30°C and 0.5 m/s < sfcWind < 17.0 m/s. Default : True.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

utci (*DataArray*) – Universal Thermal Climate Index [K] description: UTCI is the equivalent temperature for the environment derived from a reference environment and is used to evaluate heat stress in outdoor spaces.

Notes

The calculation uses water vapor partial pressure, which is derived from relative humidity and saturation vapor pressure computed according to the ITS-90 equation.

This code was inspired by the *pythermalcomfort* and *thermofeel* packages.

References

Bröde, Peter (2009). Program for calculating UTCI Temperature (UTCI), version a 0.002, http://www.utci.org/public/UTCI%20Program%20Code/UTCI_a002.f90 Błażejczyk, K., Jendritzky, G., Bröde, P., Fiala, D., Havenith, G., Epstein, Y., Psikuta, A., & Kampmann, B. (2013). An introduction to the Universal Thermal Climate Index (UTCI). DOI:10.7163/GPOL.2013.1

`xclim.indicators.atmos.warm_and_dry_days`(*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', tas_per: Union[DataArray, str] = 'tas_per', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

warm and dry days (realm: atmos)

Returns the total number of days where "warm" and "Dry" conditions coincide.

This indicator will check for missing values according to the method "from_context". Based on indice *warm_and_dry_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature values Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **tas_per** (*str or DataArray*) – Third quartile of daily mean temperature computed by month. Default : *ds.tas_per*. [Required units : [temperature]]
- **pr_per** (*str or DataArray*) – First quartile of daily total precipitation computed by month. .. warning:: Before computing the percentiles, all the precipitation below 1mm must be filtered out ! Otherwise, the percentiles will include non-wet days. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

warm_and_dry_days (*DataArray*) – warm and dry days [days] cell_methods: time: sum over days description: {freq} number of days where tas > {tas_per_thresh}th percentile and pr < {pr_per_thresh}th percentile

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [warm_dry_days].

References

`xclim.indicators.atmos.warm_and_wet_days(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', tas_per: Union[DataArray, str] = 'tas_per', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

warm and wet days (realm: atmos)

Returns the total number of days where "warm" and "wet" conditions coincide.

This indicator will check for missing values according to the method "from_context". Based on indice *warm_and_wet_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature values Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **tas_per** (*str or DataArray*) – Third quartile of daily mean temperature computed by month. Default : *ds.tas_per*. [Required units : [temperature]]
- **pr_per** (*str or DataArray*) – Third quartile of daily total precipitation computed by month. .. warning:: Before computing the percentiles, all the precipitation below 1mm must be filtered out ! Otherwise, the percentiles will include non-wet days. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

warm_and_wet_days (*DataArray*) – warm and wet days [days] cell_methods: time: sum over days description: {freq} number of days where tas > {tas_per_thresh}th percentile and pr > {pr_per_thresh}th percentile

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [warm_wet_days].

References

`xclim.indicators.atmos.warm_spell_duration_index(`*tasmax: Union[DataArray, str] = 'tasmax', tasmax_per: Union[DataArray, str] = 'tasmax_per', *, window: int = 6, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None*) → DataArray

Warm spell duration index. (realm: atmos)

Number of days inside spells of a minimum number of consecutive days where the daily maximum temperature is above the 90th percentile. The 90th percentile should be computed for a 5-day moving window, centered on each calendar day in the 1961-1990 period.

This indicator will check for missing values according to the method "from_context". Based on indice *warm_spell_duration_index()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **tasmax_per** (*str or DataArray*) – percentile(s) of daily maximum temperature. Default : *ds.tasmax_per*. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold to qualify as a warm spell. Default : 6.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

warm_spell_duration_index (*DataArray*) – Number of days part of a percentile-defined warm spell (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with at least {window} consecutive days where the daily maximum temperature is above the {tasmax_per_thresh}th percentile(s). A {tasmax_per_window} day(s) window, centred on each calendar day in the {tasmax_per_period} period, is used to compute the {tasmax_per_thresh}th percentile(s).

References

From the Expert Team on Climate Change Detection, Monitoring and Indices (ETCCDMI). Used in Alexander, L. V., et al. (2006), Global observed changes in daily climate extremes of temperature and precipitation, J. Geophys. Res., 111, D05109, doi: 10.1029/2005JD006290.

`xclim.indicators.atmos.water_budget`(*pr: Union[DataArray, str] = 'pr', evspsblpot: Optional[Union[DataArray, str]] = None, tasmin: Optional[Union[DataArray, str]] = None, tasmax: Optional[Union[DataArray, str]] = None, tas: Optional[Union[DataArray, str]] = None, lat: Optional[Union[DataArray, str]] = None, *, ds: Dataset = None*) → DataArray

Precipitation minus potential evapotranspiration. (realm: atmos)

Precipitation minus potential evapotranspiration as a measure of an approximated surface water budget, where the potential evapotranspiration can be calculated with a given method.

Based on indice `water_budget()`. With injected parameters: method=dummy.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **evspsblpot** (*str or DataArray, optional*) – Potential evapotranspiration [Required units : [precipitation]]
- **tasmin** (*str or DataArray, optional*) – Minimum daily temperature. [Required units : [temperature]]
- **tasmax** (*str or DataArray, optional*) – Maximum daily temperature. [Required units : [temperature]]
- **tas** (*str or DataArray, optional*) – Mean daily temperature. [Required units : [temperature]]
- **lat** (*str or DataArray, optional*) – Latitude, needed if evspsblpot is not given. [Required units : []]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

water_budget (*DataArray*) – Water budget [kg m-2 s-1] description: Precipitation minus potential evapotranspiration as a measure of an approximated surface water budget.

Notes

Available methods are listed in the description of xclim.indicators.atmos.potential_evapotranspiration.

`xclim.indicators.atmos.water_budget_from_tas`(*pr: Union[DataArray, str] = 'pr', evspsblpot: Optional[Union[DataArray, str]] = None, tasmin: Optional[Union[DataArray, str]] = None, tasmax: Optional[Union[DataArray, str]] = None, tas: Optional[Union[DataArray, str]] = None, lat: Optional[Union[DataArray, str]] = None, *, method: str = 'BR65', ds: Dataset = None*) → DataArray

Precipitation minus potential evapotranspiration. (realm: atmos)

Precipitation minus potential evapotranspiration as a measure of an approximated surface water budget, where the potential evapotranspiration can be calculated with a given method.

Based on indice `water_budget()`.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **evspsblpot** (*str or DataArray, optional*) – Potential evapotranspiration [Required units : [precipitation]]
- **tasmin** (*str or DataArray, optional*) – Minimum daily temperature. [Required units : [temperature]]
- **tasmax** (*str or DataArray, optional*) – Maximum daily temperature. [Required units : [temperature]]
- **tas** (*str or DataArray, optional*) – Mean daily temperature. [Required units : [temperature]]
- **lat** (*str or DataArray, optional*) – Latitude, needed if evspsblpot is not given. [Required units : []]
- **method** (*str*) – Method to use to calculate the potential evapotranspiration. Default : BR65.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

water_budget_from_tas (*DataArray*) – Water budget [kg m-2 s-1] description: Precipitation minus potential evapotranspiration as a measure of an approximated surface water budget, where the potential evapotranspiration is calculated with the method {method}.

Notes

Available methods are listed in the description of xclim.indicators.atmos.potential_evapotranspiration.

`xclim.indicators.atmos.wet_precip_accumulation`(*pr: Union[DataArray, str] = 'pr', *, thresh: str = '1 mm/day', freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Accumulated total precipitation (solid and liquid) during wet days (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice `prcptot()`.

Parameters

- **pr** (*str or DataArray*) – Total precipitation flux [mm d-1], [mm week-1], [mm month-1] or similar. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold over which precipitation starts being cumulated. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

wet_prcptot (*DataArray*) – Total precipitation (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum over days description: {freq} total precipitation over wet days, defined as days where precipitation exceeds {thresh}.

`xclim.indicators.atmos.wetdays`(*pr: Union[DataArray, str] = 'pr'*, ***, *thresh: str = '1.0 mm/day'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Wet days. (realm: atmos)

Return the total number of days during period with precipitation over threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *wetdays()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1.0 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

wetdays (*DataArray*) – Number of wet days (precip >= {thresh}) (number_of_days_with_lwe_thickness_of_precipitation_amount_at_or_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with daily precipitation over {thresh}.

`xclim.indicators.atmos.wetdays_prop`(*pr: Union[DataArray, str] = 'pr'*, ***, *thresh: str = '1.0 mm/day'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Proportion of wet days. (realm: atmos)

Return the proportion of days during period with precipitation over threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *wetdays_prop()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1.0 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

wetdays_prop (*DataArray*) – Proportion of wet days (precip >= {thresh}) [1] cell_methods: time: sum over days description: {freq} proportion of days with precipitation over {thresh}.

`xclim.indicators.atmos.wind_chill_index(`*tas: Union[DataArray, str] = 'tas', sfcWind: Union[DataArray, str] = 'sfcWind', *, method: str = 'CAN', ds: Dataset = None*`)` → DataArray

Wind chill index. (realm: atmos)

The Wind Chill Index is an estimation of how cold the weather feels to the average person. It is computed from the air temperature and the 10-m wind. As defined by the Environment and Climate Change Canada ([MVSZ2015]), two equations exist, the conventional one and one for slow winds (usually < 5 km/h), see Notes.

Based on indice *wind_chill_index()*. With injected parameters: mask_invalid=True.

Parameters

- **tas** (*str or DataArray*) – Surface air temperature. Default : *ds.tas*. [Required units : [temperature]]
- **sfcWind** (*str or DataArray*) – Surface wind speed (10 m). Default : *ds.sfcWind*. [Required units : [speed]]
- **method** (*{'US', 'CAN'}*) – If "CAN" (default), a "slow wind" equation is used where winds are slower than 5 km/h, see Notes. Default : CAN.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

wind_chill (*DataArray*) – Wind chill index [degC] description: <Dynamically generated string>

Notes

Following the calculations of Environment and Climate Change Canada, this function switches from the standardized index to another one for slow winds. The standard index is the same as used by the National Weather Service of the USA ([NWS]). Given a temperature at surface T (in °C) and 10-m wind speed V (in km/h), the Wind Chill Index W (dimensionless) is computed as:

$$W = 13.12 + 0.6125 * T - 11.37 * V^0.16 + 0.3965 * T * V^0.16$$

Under slow winds ($V < 5$ km/h), and using the canadian method, it becomes:

$$W = T + \frac{-1.59 + 0.1345 * T}{5} * V$$

Both equations are invalid for temperature over 0°C in the canadian method.

The american Wind Chill Temperature index (WCT), as defined by USA's National Weather Service, is computed when *method='US'*. In that case, the maximal valid temperature is 50°F (10 °C) and minimal wind speed is 3 mph (4.8 km/h).

References

`xclim.indicators.atmos.wind_speed_from_vector(`*uas: Union[DataArray, str] = 'uas', vas: Union[DataArray, str] = 'vas', *, calm_wind_thresh: str = '0.5 m/s', ds: Dataset = None*`)` → Tuple[DataArray, DataArray]

Wind speed and direction from the eastward and northward wind components. (realm: atmos)

Computes the magnitude and angle of the wind vector from its northward and eastward components, following the meteorological convention that sets calm wind to a direction of 0° and northerly wind to 360°.

Based on indice `uas_vas_2_sfcwind()`.

Parameters

- **uas** (*str or DataArray*) – Eastward wind velocity Default : *ds.uas*. [Required units : [speed]]
- **vas** (*str or DataArray*) – Northward wind velocity Default : *ds.vas*. [Required units : [speed]]
- **calm_wind_thresh** (*quantity (string with units)*) – The threshold under which winds are considered "calm" and for which the direction is set to 0. On the Beaufort scale, calm winds are defined as < 0.5 m/s. Default : 0.5 m/s. [Required units : [speed]]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

- **sfcWind** (*DataArray*) – Near-Surface Wind Speed (wind_speed) [m s-1] description: Wind speed computed as the magnitude of the (uas, vas) vector.
- **sfcWindfromdir** (*DataArray*) – Near-Surface Wind from Direction (wind_from_direction) [degree] description: Wind direction computed as the angle of the (uas, vas) vector. A direction of 0° is attributed to winds with a speed under {calm_wind_thresh}.

Notes

Winds with a velocity less than *calm_wind_thresh* are given a wind direction of 0°, while stronger northerly winds are set to 360°.

`xclim.indicators.atmos.wind_vector_from_speed(`*sfcWind: Union[DataArray, str] = 'sfcWind', sfcWindfromdir: Union[DataArray, str] = 'sfcWindfromdir', *, ds: Dataset = None*) → Tuple[DataArray, DataArray]

Eastward and northward wind components from the wind speed and direction. (realm: atmos)

Compute the eastward and northward wind components from the wind speed and direction.

Based on indice `sfcwind_2_uas_vas()`.

Parameters

- **sfcWind** (*str or DataArray*) – Wind velocity Default : *ds.sfcWind*. [Required units : [speed]]
- **sfcWindfromdir** (*str or DataArray*) – Direction from which the wind blows, following the meteorological convention where 360 stands for North. Default : *ds.sfcWindfromdir*. [Required units : []]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

- **uas** (*DataArray*) – Near-Surface Eastward Wind (eastward_wind) [m s-1] description: Eastward wind speed computed from its speed and direction of origin.

- **vas** (*DataArray*) – Near-Surface Northward Wind (northward_wind) [m s-1] description: Northward wind speed computed from its speed and direction of origin.

`xclim.indicators.atmos.windy_days`(*sfcWind: Union[DataArray, str] = 'sfcWind'*, ***, *thresh: str = '10.8 m s-1'*, *freq: str = 'MS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Windy days. (realm: atmos)

The number of days with average near-surface wind speed above threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *`windy_days()`*.

Parameters

- **sfcWind** (*str or DataArray*) – Daily average near-surface wind speed. Default : *ds.sfcWind*. [Required units : [speed]]
- **thresh** (*quantity (string with units)*) – Threshold average near-surface wind speed on which to base evaluation. Default : 10.8 m s-1. [Required units : [speed]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

windy_days (*DataArray*) – Number of days with surface wind speed above threshold (number_of_days_with_sfcWind_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with surface wind speed >= {thresh}

Notes

Let WS_{ij} be the windspeed at day i of period j. Then counted is the number of days where:

$$WS_{ij} >= Threshold[ms-1]$$

14.1.2 Land indicators

`xclim.indicators.land.base_flow_index`(*q: Union[DataArray, str] = 'q'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Base flow index. (realm: land)

Return the base flow index, defined as the minimum 7-day average flow divided by the mean flow.

This indicator will check for missing values according to the method "from_context". Based on indice *`base_flow_index()`*.

Parameters

- **q** (*str or DataArray*) – Rate of river discharge. Default : *ds.q*. [Required units : [discharge]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

base_flow_index (*DataArray*) – Base flow index description: Minimum 7-day average flow divided by the mean flow.

Notes

Let $\mathbf{q} = q_0, q_1, \ldots, q_n$ be the sequence of daily discharge and $\overline{\mathbf{q}}$ the mean flow over the period. The base flow index is given by:

$$\frac{\min(\mathrm{CMA}_7(\mathbf{q}))}{\overline{\mathbf{q}}}$$

where CMA_7 is the seven days moving average of the daily flow:

$$\mathrm{CMA}_7(q_i) = \frac{\sum_{j=i-3}^{i+3} q_j}{7}$$

`xclim.indicators.land.blowing_snow(`*snd: Union[DataArray, str] = 'snd', sfcWind: Union[DataArray, str] = 'sfcWind', *, snd_thresh: str = '5 cm', sfcWind_thresh: str = '15 km/h', window: int = 3, freq: str = 'AS-JUL', ds: Dataset = None*`)` → DataArray

Days with blowing snow events. (realm: land)

Number of days where both snowfall over the last days and daily wind speeds are above respective thresholds.

This indicator will check for missing values according to the method "from_context". Based on indice `blowing_snow()`.

Parameters

- **snd** (*str or DataArray*) – Surface snow depth. Default : *ds.snd*. [Required units : [length]]
- **sfcWind** (*str or DataArray*) – Wind velocity Default : *ds.sfcWind*. [Required units : [speed]]
- **snd_thresh** (*quantity (string with units)*) – Threshold on net snowfall accumulation over the last *window* days. Default : 5 cm. [Required units : [length]]
- **sfcWind_thresh** (*quantity (string with units)*) – Wind speed threshold. Default : 15 km/h. [Required units : [speed]]
- **window** (*number*) – Period over which snow is accumulated before comparing against threshold. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

{freq}_blowing_snow (*DataArray*) – Number of days where snowfall and wind speeds are above respective thresholds. [days] description: {freq} number of days with snowfall over last {window} days above {snd_thresh} and wind speed above {sfcWind_thresh}.

`xclim.indicators.land.continuous_snow_cover_end(`*snd: Union[DataArray, str] = 'snd', *, thresh: str = '2 cm', window: int = 14, freq: str = 'AS-JUL', ds: Dataset = None*`)` → DataArray

End date of continuous snow cover. (realm: land)

First day after the start of the continuous snow cover when snow depth is below *threshold* for at least *window* consecutive days. WARNING: The default *freq* is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice `continuous_snow_cover_end()`.

Parameters

- **snd** (*str or DataArray*) – Surface snow thickness. Default : *ds.snd*. [Required units : [length]]
- **thresh** (*quantity (string with units)*) – Threshold snow thickness. Default : 2 cm. [Required units : [length]]
- **window** (*number*) – Minimum number of days with snow depth below threshold. Default : 14.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

continuous_snow_cover_end (*DataArray*) – End date of continuous snow cover (day_of_year) description: Day of year when snow depth is below {thresh} for {window} consecutive days.

References

Chaumont D., Mailhot A., Diaconescu E.P., Fournier É., Logan T. 2017: Élaboration du portrait bioclimatique futur du Nunavik – Tome II. [Rapport présenté au Ministère de la forêt, de la faune et des parcs], Ouranos.

`xclim.indicators.land.continuous_snow_cover_start(`*snd: Union[DataArray, str] = 'snd', *, thresh: str = '2 cm', window: int = 14, freq: str = 'AS-JUL', ds: Dataset = None*`)` → DataArray

Start date of continuous snow cover. (realm: land)

Day of year when snow depth is above or equal *threshold* for at least *window* consecutive days. WARNING: The default *freq* is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice `continuous_snow_cover_start()`.

Parameters

- **snd** (*str or DataArray*) – Surface snow thickness. Default : *ds.snd*. [Required units : [length]]
- **thresh** (*quantity (string with units)*) – Threshold snow thickness. Default : 2 cm. [Required units : [length]]
- **window** (*number*) – Minimum number of days with snow depth above or equal to threshold. Default : 14.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

continuous_snow_cover_start (*DataArray*) – Start date of continuous snow cover (day_of_year) description: Day of year when snow depth is above or equal to {thresh} for {window} consecutive days.

References

Chaumont D., Mailhot A., Diaconescu E.P., Fournier É., Logan T. 2017: Élaboration du portrait bioclimatique futur du Nunavik – Tome II. [Rapport présenté au Ministère de la forêt, de la faune et des parcs], Ouranos.

`xclim.indicators.land.doy_qmax(`*da: Union[DataArray, str] = 'da', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Day of year of the maximum. (realm: land)

This indicator will check for missing values according to the method "from_context". Based on indice *select_resample_op()*. With injected parameters: op=<function doymax at 0x7fea13c34f70>.

Parameters

- **da** (*str or DataArray*) – Input data. Default : *ds.da*.
- **freq** (*offset alias (string)*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered. Default : None.

Returns

q{indexer}_doy_qmax (*DataArray*) – Day of the year of the maximum over {indexer} description: Day of the year of the maximum over {indexer}

`xclim.indicators.land.doy_qmin(`*da: Union[DataArray, str] = 'da', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Day of year of the minimum. (realm: land)

This indicator will check for missing values according to the method "from_context". Based on indice *select_resample_op()*. With injected parameters: op=<function doymin at 0x7fea13c10040>.

Parameters

- **da** (*str or DataArray*) – Input data. Default : *ds.da*.
- **freq** (*offset alias (string)*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered. Default : None.

Returns

q{indexer}_doy_qmin (*DataArray*) – Day of the year of the minimum over {indexer} description: Day of the year of the minimum over {indexer}

`xclim.indicators.land.fit(`*da: Union[DataArray, str] = 'da', *, dist: str = 'norm', method: str = 'ML', dim: str = 'time', ds: Dataset = None, **fitkwargs*`)` → DataArray

Distribution parameters fitted over the time dimension. (realm: land)

Based on indice *fit()*.

Parameters

- **da** (*str or DataArray*) – Time series to be fitted along the time dimension. Default : *ds.da*.
- **dist** (*str*) – Name of the univariate distribution, such as beta, expon, genextreme, gamma, gumbel_r, lognorm, norm (see scipy.stats for full list). If the PWM method is used, only the following distributions are currently supported: 'expon', 'gamma', 'genextreme', 'genpareto', 'gumbel_r', 'pearson3', 'weibull_min'. Default : norm.
- **method** (*{'PWM', 'ML'}*) – Fitting method, either maximum likelihood (ML) or probability weighted moments (PWM), also called L-Moments. The PWM method is usually more robust to outliers. Default : ML.
- **dim** (*str*) – The dimension upon which to perform the indexing (default: "time"). Other arguments passed directly to `_fitstart()` and to the distribution's *fit*. Default : time.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **fitkwargs** – Default : None.

Returns

params (*DataArray*) – {dist} distribution parameters ({dist} parameters) cell_methods: time: fit description: Parameters of the {dist} distribution

Notes

Coordinates for which all values are NaNs will be dropped before fitting the distribution. If the array still contains NaNs, the distribution parameters will be returned as NaNs.

`xclim.indicators.land.freq_analysis(`*da: Union[DataArray, str] = 'da', *, mode: str, t: int | Sequence[int], dist: str, window: int = 1, freq: str | None = None, ds: Dataset = None, **indexer*`)` → DataArray

Flow values for given return periods. (realm: land)

This indicator will check for missing values according to the method "skip". Based on indice *frequency_analysis()*.

Parameters

- **da** (*str or DataArray*) – Input data. Default : *ds.da*.
- **mode** (*{'min', 'max'}*) – Whether we are looking for a probability of exceedance (high) or a probability of non-exceedance (low). Default : *ds.da*.
- **t** (*number or sequence of numbers*) – Return period. The period depends on the resolution of the input data. If the input array's resolution is yearly, then the return period is in years. Default : *ds.da*.

- **dist** (*str*) – Name of the univariate distribution, such as *beta*, *expon*, *genextreme*, *gamma*, *gumbel_r*, *lognorm*, *norm* (see scipy.stats). Default : *ds.da*.
- **window** (*number*) – Averaging window length (days). Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. If None, the frequency is assumed to be 'YS' unless the indexer is season='DJF', in which case *freq* would be set to *AS-DEC*. Default : None.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered. Default : None.

Returns

q{window}{mode (*r}{indexer} : DataArray*) – N-year return period {mode} {indexer} {window}-day flow [m^3 s-1] description: Streamflow frequency analysis for the {mode} {indexer} {window}-day flow estimated using the {dist} distribution.

`xclim.indicators.land.rb_flashiness_index(`*q: Union[DataArray, str] = 'q', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Richards-Baker flashiness index. (realm: land)

Measures oscillations in flow relative to total flow, quantifying the frequency and rapidity of short term changes in flow, based on Baker et al. (2004; [baker2004]).

This indicator will check for missing values according to the method "from_context". Based on indice *rb_flashiness_index()*.

Parameters

- **q** (*str or DataArray*) – Rate of river discharge. Default : *ds.q*. [Required units : [discharge]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

rbi (*DataArray*) – Richards-Baker flashiness index description: {freq} R-B Index, an index measuring the flashiness of flow.

Notes

Let $\mathbf{q} = q_0, q_1, \ldots, q_n$ be the sequence of daily discharge, the R-B Index is given by:

$$\frac{\sum_{i=1}^{n} |q_i - q_{i-1}|}{\sum_{i=1}^{n} q_i}$$

References

`xclim.indicators.land.snd_max_doy`(*snd: Union[DataArray, str] = 'snd'*, ***, *freq: str = 'AS-JUL'*, *ds: Dataset = None*, ***indexer*) → DataArray

Maximum snow depth day of year. (realm: land)

Day of year when surface snow reaches its peak value. If snow depth is 0 over entire period, return NaN.

This indicator will check for missing values according to the method "from_context". Based on indice `snd_max_doy()`.

Parameters

- **snd** (*str or DataArray*) – Surface snow depth. Default : *ds.snd*. [Required units : [length]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

{freq}_snd_max_doy (*DataArray*) – Date when snow depth reaches its maximum value. (day_of_year) description: {freq} day of year when snow depth reaches its maximum value.

`xclim.indicators.land.snow_cover_duration`(*snd: Union[DataArray, str] = 'snd'*, ***, *thresh: str = '2 cm'*, *freq: str = 'AS-JUL'*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of days with snow depth above a threshold. (realm: land)

Number of days where surface snow depth is greater or equal to given threshold. WARNING: The default *freq* is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice `snow_cover_duration()`.

Parameters

- **snd** (*str or DataArray*) – Surface snow thickness. Default : *ds.snd*. [Required units : [length]]
- **thresh** (*quantity (string with units)*) – Threshold snow thickness. Default : 2 cm. [Required units : [length]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

snow_cover_duration (*DataArray*) – Number of days with snow depth above threshold [days] description: {freq} number of days with snow depth greater or equal to {thresh} `xclim.indicators.land.snow_depth`(*snd: Union[DataArray, str] = 'snd'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Mean of daily average snow depth. (realm: land)

Resample the original daily mean snow depth series by taking the mean over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *snow_depth()*.

Parameters

- **snd** (*str or DataArray*) – Default : *ds.snd*. [Required units : [length]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

snow_depth (*DataArray*) – Mean of daily snow depth (surface_snow_thickness) [cm] cell_methods: time: mean over days description: {freq} mean of daily mean snow depth.

`xclim.indicators.land.snow_melt_we_max`(*snw: Union[DataArray, str] = 'snw'*, ***, *window: int = 3*, *freq: str = 'AS-JUL'*, *ds: Dataset = None*) → DataArray

Maximum snow melt. (realm: land)

The maximum snow melt over a given number of days expressed in snow water equivalent.

This indicator will check for missing values according to the method "from_context". Based on indice *snow_melt_we_max()*.

Parameters

- **snw** (*str or DataArray*) – Snow amount (mass per area). Default : *ds.snw*. [Required units : [mass]/[area]]
- **window** (*number*) – Number of days during which the melt is accumulated. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

{freq}_snow_melt_we_max (*DataArray*) – The maximum snow melt over a given number of days for each period. [mass/area]. (change_over_time_in_surface_snow_amount) [kg m-2] description: {freq} maximum negative change in melt amount over {window} days.

`xclim.indicators.land.snw_max`(*snw: Union[DataArray, str] = 'snw'*, ***, *freq: str = 'AS-JUL'*, *ds: Dataset = None*, ***indexer*) → DataArray

Maximum snow amount. (realm: land)

The maximum daily snow amount.

This indicator will check for missing values according to the method "from_context". Based on indice *snw_max()*.

Parameters

- **snw** (*str or DataArray*) – Snow amount (mass per area). Default : *ds.snw*. [Required units : [mass]/[area]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
{freq}_snw_max (*DataArray*) – Maximum daily snow amount (surface_snow_amount) [kg m-2] description: {freq} day of year when snow amount on the surface reaches its maximum.

`xclim.indicators.land.snw_max_doy(`*snw: Union[DataArray, str] = 'snw', *, freq: str = 'AS-JUL', ds: Dataset = None, **indexer*`)` → DataArray

Maximum snow amount day of year. (realm: land)

Day of year when surface snow amount reaches its peak value. If snow amount is 0 over entire period, return NaN.

This indicator will check for missing values according to the method "from_context". Based on indice *snw_max_doy()*.

Parameters

- **snw** (*str or DataArray*) – Surface snow amount. Default : *ds.snw*. [Required units : [mass]/[area]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
{freq}_snw_max_doy (*DataArray*) – Day of year of maximum daily snow amount (day_of_year) description: {freq} maximum snow amount on the surface.

`xclim.indicators.land.stats(`*da: Union[DataArray, str] = 'da', *, op: str, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Statistic of the daily flow on a given period. (realm: land)

This indicator will check for missing values according to the method "any". Based on indice *select_resample_op()*.

Parameters

- **da** (*str or DataArray*) – Input data. Default : *ds.da*.
- **op** (*{'min', 'max', 'argmin', 'mean', 'var', 'argmax', 'sum', 'count', 'std'}*) – Reduce operation. Can either be a DataArray method or a function that can be applied to a DataArray. Default : *ds.da*.
- **freq** (*offset alias (string)*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered. Default : None.

Returns
q{indexer}{op (*r} : DataArray*) – {freq} {op} of {indexer} daily flow [m^3 s-1] description: {freq} {op} of {indexer} daily flow

`xclim.indicators.land.winter_storm(`*snd: Union[DataArray, str] = 'snd', *, thresh: str = '25 cm', freq: str = 'AS-JUL', ds: Dataset = None, **indexer*`)` → DataArray

Days with snowfall over threshold. (realm: land)

Number of days with snowfall accumulation greater or equal to threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *winter_storm()*.

Parameters

- **snd** (*str or DataArray*) – Surface snow depth. Default : *ds.snd*. [Required units : [length]]
- **thresh** (*quantity (string with units)*) – Threshold on snowfall accumulation require to label an event a *winter storm*. Default : 25 cm. [Required units : [length]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
{freq}_winter_storm (*DataArray*) – Number of days per period identified as winter storms. [days] description: {freq} number of days with snowfall accumulation above {thresh}.

Notes

Snowfall accumulation is estimated by the change in snow depth.

14.1.3 Ice-related indicators

`xclim.indicators.seaIce.sea_ice_area(`*siconc: Union[DataArray, str] = 'siconc', areacello: Union[DataArray, str] = 'areacello', *, thresh: str = '15 pct', ds: Dataset = None*`)` → DataArray

Total sea ice area. (realm: seaIce)

Sea ice area measures the total sea ice covered area where sea ice concentration is above a threshold, usually set to 15%.

This indicator will check for missing values according to the method "skip". Based on indice *sea_ice_area()*.

Parameters

- **siconc** (*str or DataArray*) – Sea ice concentration (area fraction). Default : *ds.siconc*. [Required units : []]
- **areacello** (*str or DataArray*) – Grid cell area (usually over the ocean). Default : *ds.areacello*. [Required units : [area]]
- **thresh** (*quantity (string with units)*) – Minimum sea ice concentration for a grid cell to contribute to the sea ice extent. Default : 15 pct. [Required units : []]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

sea_ice_area (*DataArray*) – Sea ice area (sea_ice_area) [m2] cell_methods: lon: sum lat: sum description: The sum of ice-covered areas where sea ice concentration is at least {thresh}.

Notes

To compute sea ice area over a subregion, first mask or subset the input sea ice concentration data.

References

What is the difference between sea ice area and extent

`xclim.indicators.seaIce.sea_ice_extent(`*siconc: Union[DataArray, str] = 'siconc', areacello: Union[DataArray, str] = 'areacello', *, thresh: str = '15 pct', ds: Dataset = None*`)` → DataArray

Total sea ice extent. (realm: seaIce)

Sea ice extent measures the *ice-covered* area, where a region is considered ice-covered if its sea ice concentration is above a threshold usually set to 15%.

This indicator will check for missing values according to the method "skip". Based on indice `sea_ice_extent()`.

Parameters

- **siconc** (*str or DataArray*) – Sea ice concentration (area fraction). Default : *ds.siconc*. [Required units : []]
- **areacello** (*str or DataArray*) – Grid cell area. Default : *ds.areacello*. [Required units : [area]]
- **thresh** (*quantity (string with units)*) – Minimum sea ice concentration for a grid cell to contribute to the sea ice extent. Default : 15 pct. [Required units : []]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

sea_ice_extent (*DataArray*) – Sea ice extent (sea_ice_extent) [m2] cell_methods: lon: sum lat: sum description: The sum of ocean areas where sea ice concentration is at least {thresh}.

Notes

To compute sea ice area over a subregion, first mask or subset the input sea ice concentration data.

References

What is the difference between sea ice area and extent

14.1.4 Virtual indicator submodules

CF Standard indices

Indicator found here are defined by the team at clix-meta. Adapted documentation from that repository follows:

The repository aims to provide a platform for thinking about, and developing, a unified view of metadata elements required to describe climate indices (aka climate indicators).

To facilitate data exchange and dissemination the metadata should, as far as possible, follow the Climate and Forecasting (CF) Conventions. Considering the very rich and diverse flora of climate indices this is however not always possible. By collecting a wide range of different indices it is easier to discover any common patterns and features that are currently not well covered by the CF Conventions. Currently identified issues frequently relate to standard_name or/and cell_methods which both are controlled vocabularies of the CF Conventions.

`xclim.indicators.cf.cdd`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: threshold=1 mm day-1, condition=<, reducer=max.

Parameters

- **pr** (*str or DataArray*) – Surface precipitation flux (all phases). Default : *ds.pr*. [Required units : kg m-2 s-1]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

cdd (*DataArray*) – Maximum consecutive dry days (Precip < 1mm) (spell_length_of_days_with_lwe_thickness_of_precipitation_amount_below_threshold) [day] cell_methods: time: sum over days proposed_standard_name: spell_length_with_lwe_thickness_of_precipitation_amount_below_threshold

References

ETCCDI clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.cddcoldTT`(*tas: Union[DataArray, str] = 'tas'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate the temperature sum above/below a threshold. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Finally, the sum is calculated for those data values that fulfill the condition after subtraction of the threshold value. If the sum is for values below the threshold the result is multiplied by -1.

This indicator will check for missing values according to the method "from_context". Based on indice *temperature_sum()*. With injected parameters: condition=>.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tas*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
cddcold{threshold} (*DataArray*) – Cooling Degree Days (Tmean > {threshold}C) (integral_wrt_time_of_air_temperature_excess) [degree_Celsius day] cell_methods: time: sum over days

References

ET-SCI clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.cfd`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate the number of times some condition is met. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, then this counts the number of times *data < threshold*. Finally, count the number of occurrences when condition is met.

This indicator will check for missing values according to the method "from_context". Based on indice *count_occurrences()*. With injected parameters: threshold=0 degree_Celsius, condition=<.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
cfd (*DataArray*) – Maximum number of consecutive frost days (Tmin < 0 C) (spell_length_of_days_with_air_temperature_below_threshold)

[day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_below_threshold

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.csu`(*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate the number of times some condition is met. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, then this counts the number of times *data < threshold*. Finally, count the number of occurrences when condition is met.

This indicator will check for missing values according to the method "from_context". Based on indice `count_occurrences()`. With injected parameters: threshold=25 degree_Celsius, condition=>.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

csu (*DataArray*) – Maximum number of consecutive summer days (Tmax >25 C) (spell_length_of_days_with_air_temperature_above_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_above_threshold

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ctmgeTT`(*tas: Union[DataArray, str] = 'tas'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: condition=>, reducer=max.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tas*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
ctmge{threshold} (*DataArray*) – Maximum number of consequtive days with Tmean >= {threshold}C (spell_length_of_days_with_air_temperature_above_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_at_or_above_threshold

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ctmgtTT`(*tas: Union[DataArray, str] = 'tas'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: condition=>, reducer=max.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tas*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
ctmgt{threshold} (*DataArray*) – Maximum number of consequtive days with Tmean > {threshold}C (spell_length_of_days_with_air_temperature_above_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_above_threshold

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ctmleTT`(*tas: Union[DataArray, str] = 'tas'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: condition=<, reducer=max.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]

- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tas*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

ctmle{threshold} (*DataArray*) – Maximum number of consequtive days with Tmean <= {threshold}C (spell_length_of_days_with_air_temperature_below_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_at_or_below_threshold

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ctmltTT(`*tas: Union[DataArray, str] = 'tas', *, threshold: str, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: condition=<, reducer=max.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tas*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

ctmlt{threshold} (*DataArray*) – Maximum number of consequtive days with Tmean < {threshold}C (spell_length_of_days_with_air_temperature_below_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_below_threshold

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ctngeTT(`*tasmin: Union[DataArray, str] = 'tasmin', *, threshold: str, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: condition=>, reducer=max.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tasmin*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

ctnge{threshold} (*DataArray*) – Maximum number of consequtive days with Tmin >= {threshold}C (spell_length_of_days_with_air_temperature_above_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_at_or_above_threshold

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ctngtTT(`*tasmin: Union[DataArray, str] = 'tasmin', *, threshold: str, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: condition=>, reducer=max.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tasmin*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

ctngt{threshold} (*DataArray*) – Maximum number of consequtive days with Tmin > {threshold}C (spell_length_of_days_with_air_temperature_above_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_above_threshold

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ctnleTT`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: condition=<, reducer=max.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tasmin*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

ctnle{threshold} (*DataArray*) – Maximum number of consequtive days with Tmin <= {threshold}C (spell_length_of_days_with_air_temperature_below_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_at_or_below_threshold

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ctnltTT`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: condition=<, reducer=max.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tasmin*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

ctnlt{threshold} (*DataArray*) – Maximum number of consequtive days with Tmin < {threshold}C (spell_length_of_days_with_air_temperature_below_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_below_threshold

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ctxgeTT(`*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*`)` → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: condition=>, reducer=max.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tasmax*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

ctxge{threshold} (*DataArray*) – Maximum number of consequtive days with Tmax >= {threshold}C (spell_length_of_days_with_air_temperature_above_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_at_or_above_threshold

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ctxgtTT(`*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*`)` → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: condition=>, reducer=max.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]

- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tasmax*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

ctxgt{threshold} (*DataArray*) – Maximum number of consequtive days with Tmax > {threshold}C (spell_length_of_days_with_air_temperature_above_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_above_threshold

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ctxleTT`(*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: condition=<, reducer=max.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tasmax*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

ctxle{threshold} (*DataArray*) – Maximum number of consequtive days with Tmax <= {threshold}C (spell_length_of_days_with_air_temperature_below_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_at_or_below_threshold

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ctxltTT`(*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: condition=<, reducer=max.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tasmax*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

ctxlt{threshold} (*DataArray*) – Maximum number of consequtive days with Tmax < {threshold}C (spell_length_of_days_with_air_temperature_below_threshold) [day] cell_methods: time: maximum over days proposed_standard_name: spell_length_with_air_temperature_below_threshold

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.cwd`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate statistics on lengths of spells. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Then the spells are determined, and finally the statistics according to the specified reducer are calculated.

This indicator will check for missing values according to the method "from_context". Based on indice `spell_length()`. With injected parameters: threshold=1 mm day-1, condition=>, reducer=max.

Parameters

- **pr** (*str or DataArray*) – Surface precipitation flux (all phases). Default : *ds.pr*. [Required units : kg m-2 s-1]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

cwd (*DataArray*) – Maximum consecutive wet days (Precip >= 1mm) (spell_length_of_days_with_lwe_thickness_of_precipitation_amount_above_threshold) [day] cell_methods: time: sum over days proposed_standard_name: spell_length_with_lwe_thickness_of_precipitation_amount_at_or_above_threshold

References

ETCCDI clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ddgtTT`(*tas: Union[DataArray, str] = 'tas'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate the temperature sum above/below a threshold. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold.

Finally, the sum is calculated for those data values that fulfill the condition after subtraction of the threshold value. If the sum is for values below the threshold the result is multiplied by -1.

This indicator will check for missing values according to the method "from_context". Based on indice *temperature_sum()*. With injected parameters: condition=>.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tas*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

ddgt{threshold} (*DataArray*) – Degree Days (Tmean > {threshold}C) (integral_wrt_time_of_air_temperature_excess) [degree_Celsius day] cell_methods: time: sum over days

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ddltTT`(*tas: Union[DataArray, str] = 'tas'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate the temperature sum above/below a threshold. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Finally, the sum is calculated for those data values that fulfill the condition after subtraction of the threshold value. If the sum is for values below the threshold the result is multiplied by -1.

This indicator will check for missing values according to the method "from_context". Based on indice *temperature_sum()*. With injected parameters: condition=<.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tas*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

ddlt{threshold} (*DataArray*) – Degree Days (Tmean < {threshold}C) (integral_wrt_time_of_air_temperature_deficit) [degree_Celsius day] cell_methods: time: sum over days

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.dtr(`*tasmax: Union[DataArray, str] = 'tasmax', tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'MS', ds: Dataset = None*`)` → DataArray

Calculate the diurnal temperature range and reduce according to a statistic. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *`diurnal_temperature_range()`*. With injected parameters: reducer=mean.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

dtr (*DataArray*) – Mean Diurnal Temperature Range [degree_Celsius] cell_methods: time: range within days time: mean over days proposed_standard_name: air_temperature_range

References

ETCCDI clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.etr(`*tasmax: Union[DataArray, str] = 'tasmax', tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'MS', ds: Dataset = None*`)` → DataArray

Calculate the extreme temperature range as the maximum of daily maximum temperature minus the minimum of daily minimum temperature. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice `extreme_temperature_range()`.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

etr (*DataArray*) – Intra-period extreme temperature range [degree_Celsius] cell_methods: time: range proposed_standard_name: air_temperature_range

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.fg(`*sfcWind: Union[DataArray, str] = 'sfcWind', *, freq: str = 'MS', ds: Dataset = None*`)` → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=mean.

Parameters

- **sfcWind** (*str or DataArray*) – Surface wind speed. Default : *ds.sfcWind*. [Required units : m s-1]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

fg (*DataArray*) – Mean of daily mean wind strength (wind_speed) [meter second-1] cell_methods: time: mean

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.fxx(`*wsgsmax: Union[DataArray, str] = 'wsgsmax', *, freq: str = 'MS', ds: Dataset = None*`)` → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=max.

Parameters

- **wsgsmax** (*str or DataArray*) – Maximum surface wind speed. Default : *ds.wsgsmax*. [Required units : m s-1]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

fxx (*DataArray*) – Maximum value of daily maximum wind gust strength (wind_speed_of_gust) [meter second-1] cell_methods: time: maximum

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.gd4(`*tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Calculate the temperature sum above/below a threshold. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Finally, the sum is calculated for those data values that fulfill the condition after subtraction of the threshold value. If the sum is for values below the threshold the result is multiplied by -1.

This indicator will check for missing values according to the method "from_context". Based on indice *temperature_sum()*. With injected parameters: threshold=4 degree_Celsius, condition=>.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

gd4 (*DataArray*) – Growing degree days (sum of Tmean > 4 C) (integral_wrt_time_of_air_temperature_excess) [degree_Celsius day] cell_methods: time: sum over days

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.gddgrowTT`(*tas: Union[DataArray, str] = 'tas'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate the temperature sum above/below a threshold. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Finally, the sum is calculated for those data values that fulfill the condition after subtraction of the threshold value. If the sum is for values below the threshold the result is multiplied by -1.

This indicator will check for missing values according to the method "from_context". Based on indice *temperature_sum()*. With injected parameters: condition=>.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tas*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

gddgrow{threshold} (*DataArray*) – Annual Growing Degree Days (Tmean > {threshold}C) (integral_wrt_time_of_air_temperature_excess) [degree_Celsius day] cell_methods: time: sum over days

References

ET-SCI clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.hd17`(*tas: Union[DataArray, str] = 'tas'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate the temperature sum above/below a threshold. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Finally, the sum is calculated for those data values that fulfill the condition after subtraction of the threshold value. If the sum is for values below the threshold the result is multiplied by -1.

This indicator will check for missing values according to the method "from_context". Based on indice `temperature_sum()`. With injected parameters: threshold=17 degree_Celsius, condition=<.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

hd17 (*DataArray*) – Heating degree days (sum of Tmean < 17 C) (integral_wrt_time_of_air_temperature_excess) [degree_Celsius day] cell_methods: time: sum over days

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.hddheatTT`(*tas: Union[DataArray, str] = 'tas'*, ***, *threshold: str*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate the temperature sum above/below a threshold. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Finally, the sum is calculated for those data values that fulfill the condition after subtraction of the threshold value. If the sum is for values below the threshold the result is multiplied by -1.

This indicator will check for missing values according to the method "from_context". Based on indice `temperature_sum()`. With injected parameters: condition=<.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **threshold** (*quantity (string with units)*) – air temperature Default : *ds.tas*. [Required units : degree_Celsius]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

hddheat{threshold} (*DataArray*) – Heating Degree Days (Tmean < {threshold}C) (integral_wrt_time_of_air_temperature_deficit) [degree_Celsius day] cell_methods: time: sum over days

References

ET-SCI clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.iter_indicators()`

Iterate over the (name, indicator) pairs in the cf indicator module.

`xclim.indicators.cf.maxdtr(`*tasmax: Union[DataArray, str] = 'tasmax', tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'MS', ds: Dataset = None*`)` → DataArray

Calculate the diurnal temperature range and reduce according to a statistic. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *diurnal_temperature_range()*. With injected parameters: reducer=max.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

maxdtr (*DataArray*) – Maximum Diurnal Temperature Range [degree_Celsius] cell_methods: time: range within days time: maximum over days proposed_standard_name: air_temperature_range

References

SMHI clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.pp(`*psl: Union[DataArray, str] = 'psl', *, freq: str = 'MS', ds: Dataset = None*`)` → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=mean.

Parameters

- **psl** (*str or DataArray*) – Air pressure at sea level. Default : *ds.psl*. [Required units : Pa]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

pp (*DataArray*) – Mean of daily sea level pressure (air_pressure_at_sea_level) [hPa] cell_methods: time: mean

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.rh`(*hurs: Union[DataArray, str] = 'hurs'*, ***, *freq: str = 'MS'*, *ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=mean.

Parameters

- **hurs** (*str or DataArray*) – Relative humidity. Default : *ds.hurs*. [Required units : %]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

rh (*DataArray*) – Mean of daily relative humidity (relative_humidity) [%] cell_methods: time: mean

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.sd`(*snd: Union[DataArray, str] = 'snd'*, ***, *freq: str = 'MS'*, *ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=mean.

Parameters

- **snd** (*str or DataArray*) – Surface snow thickness. Default : *ds.snd*. [Required units : m]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

sd (*DataArray*) – Mean of daily snow depth (surface_snow_thickness) [cm] cell_methods: time: mean

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.sdii`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data for which some condition is met. (realm: atmos)

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Finally, the statistic is calculated for those data values that fulfill the condition.

This indicator will check for missing values according to the method "from_context". Based on indice *thresholded_statistics()*. With injected parameters: threshold=1 mm day-1, condition=>, reducer=mean.

Parameters

- **pr** (*str or DataArray*) – Surface precipitation flux (all phases). Default : *ds.pr*. [Required units : kg m-2 s-1]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
sdii (*DataArray*) – Average precipitation during Wet Days (SDII) (lwe_precipitation_rate) [mm day-1] cell_methods: time: mean over days

References

ETCCDI clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.ss`(*sund: Union[DataArray, str] = 'sund', *, freq: str = 'MS', ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=sum.

Parameters

- **sund** (*str or DataArray*) – Duration of sunshine. Default : *ds.sund*. [Required units : s]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
ss (*DataArray*) – Sunshine duration, sum (duration_of_sunshine) [hour]

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tg`(*tas: Union[DataArray, str] = 'tas', *, freq: str = 'MS', ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=mean.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
tg (*DataArray*) – Mean of daily mean temperature (air_temperature) [degree_Celsius] cell_methods: time: mean

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tmm`(*tas: Union[DataArray, str] = 'tas'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=mean.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
tmm (*DataArray*) – Mean daily mean temperature (air_temperature) [degree_Celsius] cell_methods: time: mean over days

References

clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tmmax`(*tas: Union[DataArray, str] = 'tas'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=max.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
tmmax (*DataArray*) – Maximum daily mean temperature (air_temperature) [degree_Celsius] cell_methods: time: maximum over days

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tmmean(`*tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *`statistics()`*. With injected parameters: reducer=mean.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

tmmean (*DataArray*) – Mean daily mean temperature (air_temperature) [degree_Celsius] cell_methods: time: mean over days

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tmmin(`*tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *`statistics()`*. With injected parameters: reducer=min.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

tmmin (*DataArray*) – Minimum daily mean temperature (air_temperature) [degree_Celsius] cell_methods: time: maximum over days

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tmn(`*tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *`statistics()`*. With injected parameters: reducer=min.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
tmn (*DataArray*) – Minimum daily mean temperature (air_temperature) [degree_Celsius] cell_methods: time: minimum over days

References

clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tmx`(*tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice `statistics()`. With injected parameters: reducer=max.

Parameters

- **tas** (*str or DataArray*) – Mean surface temperature. Default : *ds.tas*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
tmx (*DataArray*) – Maximum daily mean temperature (air_temperature) [degree_Celsius] cell_methods: time: maximum over days

References

clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tn`(*tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'MS', ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice `statistics()`. With injected parameters: reducer=mean.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
tn (*DataArray*) – Mean of daily minimum temperature (air_temperature) [degree_Celsius] cell_methods: time: mean

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tnm`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=mean.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

tnm (*DataArray*) – Mean daily minimum temperature (air_temperature) [degree_Celsius] cell_methods: time: mean over days

References

clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tnmax`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=max.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

tnmax (*DataArray*) – Maximum daily minimum temperature (air_temperature) [degree_Celsius] cell_methods: time: maximum over days

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tnmean`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=mean.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
tnmean (*DataArray*) – Mean daily minimum temperature (air_temperature) [degree_Celsius] cell_methods: time: mean over days

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tnmin`(*tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'YS', ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=min.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
tnmin (*DataArray*) – Minimum daily minimum temperature (air_temperature) [degree_Celsius] cell_methods: time: minimum over days

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tnn`(*tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'YS', ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=min.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
tnn (*DataArray*) – Minimum daily minimum temperature (air_temperature) [degree_Celsius] cell_methods: time: minimum over days

References

ETCCDI clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tnx`(*tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'YS', ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *`statistics()`*. With injected parameters: reducer=max.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

tnx (*DataArray*) – Maximum daily minimum temperature (air_temperature) [degree_Celsius] cell_methods: time: maximum over days

References

ETCCDI clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.tx`(*tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'MS', ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *`statistics()`*. With injected parameters: reducer=mean.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

tx (*DataArray*) – Mean of daily maximum temperature (air_temperature) [degree_Celsius] cell_methods: time: mean

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.txm`(*tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *`statistics()`*. With injected parameters: reducer=mean.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
txm (*DataArray*) – Mean daily maximum temperature (air_temperature) [degree_Celsius] cell_methods: time: mean over days

References

clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.txmax`(*tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=max.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
txmax (*DataArray*) – Maximum daily maximum temperature (air_temperature) [degree_Celsius] cell_methods: time: maximum over days

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.txmean`(*tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=mean.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
txmean (*DataArray*) – Mean daily maximum temperature (air_temperature) [degree_Celsius] cell_methods: time: mean over days

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.txmin`(*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=min.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

txmin (*DataArray*) – Minimum daily maximum temperature (air_temperature) [degree_Celsius] cell_methods: time: minimum over days

References

CLIPC clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.txn`(*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=min.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

txn (*DataArray*) – Minimum daily maximum temperature (air_temperature) [degree_Celsius] cell_methods: time: minimum over days

References

ETCCDI clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.txx`(*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Calculate a simple statistic of the data. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *statistics()*. With injected parameters: reducer=max.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
txx (*DataArray*) – Maximum daily maximum temperature (air_temperature) [degree_Celsius] cell_methods: time: maximum over days

References

ETCCDI clix-meta https://github.com/clix-meta/clix-meta

`xclim.indicators.cf.vdtr(`*tasmax: Union[DataArray, str] = 'tasmax', tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'MS', ds: Dataset = None*`)` → DataArray

Calculate the average absolute day-to-day difference in diurnal temperature range. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *interday_diurnal_temperature_range()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
vdtr (*DataArray*) – Mean day-to-day variation in Diurnal Temperature Range [degree_Celsius] proposed_standard_name: air_temperature_difference

References

ECA&D clix-meta https://github.com/clix-meta/clix-meta

ICCLIM indices

The European Climate Assessment & Dataset project (**`ECAD`** _) defines a set of 26 core climate indices. Those have been made accessible directly in xclim through their ECAD name for compatibility. However, the methods in this module are only wrappers around the corresponding methods of *xclim.indices*. Note that none of the checks performed by the *xclim.utils.Indicator* class (like with *xclim.atmos* indicators) are performed in this module.

`xclim.indicators.icclim.BEDD(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Biologically effective growing degree days. (realm: atmos)

Growing-degree days with a base of 10°C and an upper limit of 19°C and adjusted for latitudes between 40°N and 50°N for April to October (Northern Hemisphere; October to April in Southern Hemisphere). A temperature range adjustment also promotes small and large swings in daily temperature range. Used as a heat-summation metric in viticulture agroclimatology.

This indicator will check for missing values according to the method "from_context". Based on indice `biologically_effective_degree_days()`. With injected parameters: lat=None, thresh_tasmin=10 degC, method=icclim, low_dtr=None, high_dtr=None, max_daily_degree_days=9 degC, start_date=04-01, end_date=10-01.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency (default: "YS"; For Southern Hemisphere, should be "AS-JUL"). Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

BEDD (*DataArray*) – Biologically effective growing degree days (Summation of min((max((Tmin + Tmax)/2 - {thresh_tasmin}, 0)), 9°C), for days between 1 April and 30 September) [K days] comment: Revised formula published by ECAD/KNMI for ICCLIM, 2013. description: Heat-summation index for agroclimatic suitability estimation, developed specifically for viticulture. Considers daily Tmin and Tmax with a base of {thresh_tasmin} between 1 April and 31 October, with a maximum daily value for degree days (typically 9°C). It also integrates a modification coefficient for latitudes between 40°N and 50°N as well as swings in daily temperature range.

Notes

The tasmax ceiling of 19°C is assumed to be the max temperature beyond which no further gains from daily temperature occur. Indice originally published in [Gladstones1992].

Let TX_i and TN_i be the daily maximum and minimum temperature at day i, lat the latitude of the point of interest, $degdays_{max}$ the maximum amount of degrees that can be summed per day (typically, 9). Then the sum of daily biologically effective growing degree day (BEDD) units between 1 April and 31 October is:

$$BEDD_i = \sum_{i=\text{April 1}}^{\text{October 31}} min\left(\left(max\left(\frac{TX_i + TN_i)}{2} - 10, 0\right) * k\right) + TR_{adj}, degdays_{max}\right)$$

$$TR_{adj} = f(TX_i, TN_i) = \begin{cases} 0.25(TX_i - TN_i - 13), & \text{if } (TX_i - TN_i) > 13 \\ 0, & \text{if } 10 < (TX_i - TN_i) < 13 \\ 0.25(TX_i - TN_i - 10), & \text{if } (TX_i - TN_i) < 10 \end{cases}$$

$$k = f(lat) = 1 + \left(\frac{|lat|}{50} * 0.06, \text{if } 40 < |lat| < 50, \text{else } 0\right)$$

A second version of the BEDD (*method="icclim"*) does not consider TR_{adj} and k and employs a different end date (30 September) ([ECAD]). The simplified formula is as follows:

$$BEDD_i = \sum_{i=\text{April 1}}^{\text{September 30}} min\left(max\left(\frac{TX_i + TN_i)}{2} - 10, 0\right), degdays_{max}\right)$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.CD(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', tas_per: Union[DataArray, str] = 'tas_per', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Cold and dry days (realm: atmos)

Returns the total number of days where "Cold" and "Dry" conditions coincide.

This indicator will check for missing values according to the method "from_context". Based on indice *cold_and_dry_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature values Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **tas_per** (*str or DataArray*) – Daily 25th percentile of temperature. Default : *ds.tas_per*. [Required units : [temperature]]
- **pr_per** (*str or DataArray*) – Daily 25th percentile of wet day precipitation flux. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

CD (*DataArray*) – Cold and dry days [days] cell_methods: time: sum over days description: {freq} number of days where tas < {tas_per_thresh}th percentile and pr < {pr_per_thresh}th percentile

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [cold_dry_days].

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.CDD`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Maximum number of consecutive dry days. (realm: atmos)

Return the maximum number of consecutive days within the period where precipitation is below a certain threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *maximum_consecutive_dry_days()*. With injected parameters: thresh=1 mm/day.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

CDD (*DataArray*) – Maximum number of consecutive dry days (RR<1 mm) (number_of_days_with_lwe_thickness_of_precipitation_amount_below_threshold) [days] cell_methods: time: sum over days description: {freq} maximum number of consecutive days with daily precipitation below {thresh}.

Notes

Let $\mathbf{p} = p_0, p_1, \ldots, p_n$ be a daily precipitation series and *thresh* the threshold under which a day is considered dry. Then let $\mathbf{s}$ be the sorted vector of indices i where $[p_i < thresh] \neq [p_{i+1} < thresh]$, that is, the days when the precipitation crosses the threshold. Then the maximum number of consecutive dry days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[p_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.CFD`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *freq: str = 'AS-JUL'*, *ds: Dataset = None*) → DataArray

Maximum number of consecutive frost days (Tn < 0℃). (realm: atmos)

The maximum number of consecutive days within the period where the temperature is under a certain threshold (default: 0°C). WARNING: The default freq value is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *maximum_consecutive_frost_days()*. With injected parameters: thresh=0 degC.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]

- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
CFD (*DataArray*) – Maximum number of consecutive frost days (TN<0°C) (spell_length_of_days_with_air_temperature_below_threshold) [days] cell_methods: time: maximum over days description: {freq} maximum number of consecutive days with minimum daily temperature below {thresh}.

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily minimum temperature series and *thresh* the threshold below which a day is considered a frost day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i < thresh] \neq [t_{i+1} < thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive frost free days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.CSDI(`*tasmin: Union[DataArray, str] = 'tasmin', tasmin_per: Union[DataArray, str] = 'tasmin_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None*`)` → DataArray

Cold spell duration index. (realm: atmos)

Number of days with at least *window* consecutive days where the daily minimum temperature is below the *tasmin_per* percentiles.

This indicator will check for missing values according to the method "from_context". Based on indice `cold_spell_duration_index()`. With injected parameters: window=6.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmin_per** (*str or DataArray*) – nth percentile of daily minimum temperature with *dayofyear* coordinate. Default : *ds.tasmin_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

CSDI (*DataArray*) – Cold-spell duration index (cold_spell_duration_index) [days] description: {freq} number of days with at least {window} consecutive days where the daily minimum temperature is below the {tasmin_per_thresh}th percentile(s). A {tasmin_per_window} day(s) window, centred on each calendar day in the {tasmin_per_period} period, is used to compute the {tasmin_per_thresh}th percentile(s).

Notes

Let TN_i be the minimum daily temperature for the day of the year i and $TN10_i$ the 10th percentile of the minimum daily temperature over the 1961-1990 period for day of the year i, the cold spell duration index over period ϕ is defined as:

$$\sum_{i \in \phi} \prod_{j=i}^{i+6} [TN_j < TN10_j]$$

where $[P]$ is 1 if P is true, and 0 if false.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.CSU(`*tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Maximum number of consecutive days with tasmax above a threshold (summer days). (realm: atmos)

Return the maximum number of consecutive days within the period where the maximum temperature is above a certain threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *maximum_consecutive_tx_days()*. With injected parameters: thresh=25 degC.

Parameters

- **tasmax** (*str or DataArray*) – Max daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

CSU (*DataArray*) – Maximum number of consecutive summer day (spell_length_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: maximum over days description: {freq} longest spell of consecutive days with Tmax above {thresh}.

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily maximum temperature series and $thresh$ the threshold above which a day is considered a summer day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i < thresh] \neq [t_{i+1} < thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive dry days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.CW(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', tas_per: Union[DataArray, str] = 'tas_per', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

cold and wet days (realm: atmos)

Returns the total number of days where “cold” and “wet” conditions coincide.

This indicator will check for missing values according to the method “from_context”. Based on indice *cold_and_wet_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature values Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **tas_per** (*str or DataArray*) – Daily 25th percentile of temperature. Default : *ds.tas_per*. [Required units : [temperature]]
- **pr_per** (*str or DataArray*) – Daily 75th percentile of wet day precipitation flux. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

CW (*DataArray*) – cold and wet days [days] cell_methods: time: sum over days description: {freq} number of days where tas < {tas_per_thresh}th percentile and pr > {pr_per_thresh}th percentile

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [cold_wet_days].

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.CWD(`*pr: Union[DataArray, str] = 'pr', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Consecutive wet days. (realm: atmos)

Returns the maximum number of consecutive wet days.

This indicator will check for missing values according to the method "from_context". Based on indice *`maximum_consecutive_wet_days()`*. With injected parameters: thresh=1 mm/day.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

CWD (*DataArray*) – Maximum number of consecutive wet days (RR1 mm) (number_of_days_with_lwe_thickness_of_precipitation_amount_at_or_above_threshold) [days] cell_methods: time: sum over days description: {freq} maximum number of consecutive days with daily precipitation over {thresh}.

Notes

Let $\mathbf{x} = x_0, x_1, \ldots, x_n$ be a daily precipitation series and $\mathbf{s}$ be the sorted vector of indices i where $[p_i > thresh] \neq [p_{i+1} > thresh]$, that is, the days when the precipitation crosses the *wet day* threshold. Then the maximum number of consecutive wet days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[x_{s_j} > 0^\circ C]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.DTR(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Mean of daily temperature range. (realm: atmos)

The mean difference between the daily maximum temperature and the daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *daily_temperature_range()*. With injected parameters: op=mean.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

DTR (*DataArray*) – Mean of diurnal temperature range (air_temperature) [K] cell_methods: time range within days time: mean over days description: {freq} mean diurnal temperature range.

Notes

For a default calculation using *op='mean'* :

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the mean diurnal temperature range in period j is:

$$DTR_j = \frac{\sum_{i=1}^{I}(TX_{ij} - TN_{ij})}{I}$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.ETR`(*tasmin: Union[DataArray, str] = 'tasmin'*, *tasmax: Union[DataArray, str] = 'tasmax'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Extreme intra-period temperature range. (realm: atmos)

The maximum of max temperature (TXx) minus the minimum of min temperature (TNn) for the given time period.

This indicator will check for missing values according to the method "from_context". Based on indice *extreme_temperature_range()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

ETR (*DataArray*) – Intra-period extreme temperature range (air_temperature) [K] description: {freq} range between the maximum of daily max temperature (tx_max) and the minimum of daily min temperature (tn_min)

Notes

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the extreme temperature range in period j is:

$$ETR_j = max(TX_{ij}) - min(TN_{ij})$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.FD`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Frost days index. (realm: atmos)

Number of days where daily minimum temperatures are below a threshold temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *`frost_days()`*. With injected parameters: thresh=0 degC.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

FD (*DataArray*) – Frost days (TN<0°C) (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with minimum daily temperature below {thresh}.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j and :math`TT` the threshold. Then counted is the number of days where:

$$TN_{ij} < TT$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.GD4`(*tas: Union[DataArray, str] = 'tas'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Growing degree-days over threshold temperature value. (realm: atmos)

The sum of degree-days over the threshold temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *growing_degree_days()*. With injected parameters: thresh=4 degC.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

GD4 (*DataArray*) – Growing degree days (sum of TG>4°C) (integral_of_air_temperature_excess_wrt_time) [K days] cell_methods: time: sum over days description: {freq} growing degree days above {thresh}.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then the growing degree days are:

$$GD4_j = \sum_{i=1}^{I}(TG_{ij} - 4 | TG_{ij} > 4)$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.GSL`(*tas: Union[DataArray, str] = 'tas'*, ***, *mid_date: DayOfYearStr = '07-01'*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Growing season length. (realm: atmos)

The number of days between the first occurrence of at least six consecutive days with mean daily temperature over a threshold (default: 5°C) and the first occurrence of at least six consecutive days with mean daily temperature below the same threshold after a certain date. (Usually July 1st in the northern emisphere and January 1st in the southern hemisphere.)

This indicator will check for missing values according to the method "from_context". Based on indice *growing_season_length()*. With injected parameters: thresh=5 degC, window=6.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]

- **mid_date** (*date (string, MM-DD)*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'. Default : 07-01.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

GSL (*DataArray*) – Growing season length (growing_season_length) [days] description: {freq} number of days between the first occurrence of at least {window} consecutive days with mean daily temperature over {thresh} and the first occurrence of at least {window} consecutive days with mean daily temperature below {thresh} after {mid_date}.

Notes

Let TG_{ij} be the mean temperature at day i of period j. Then counted is the number of days between the first occurrence of at least 6 consecutive days with:

$$TG_{ij} > 5$$

and the first occurrence after 1 July of at least 6 consecutive days with:

$$TG_{ij} < 5$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.HD17(`*tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Heating degree days. (realm: atmos)

Sum of degree days below the temperature threshold at which spaces are heated.

This indicator will check for missing values according to the method "from_context". Based on indice *heating_degree_days()*. With injected parameters: thresh=17 degC.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

HD17 (*DataArray*) – Heating degree days (sum of17°C - TG) (integral_of_air_temperature_deficit_wrt_time) [K days] cell_methods: time: sum over days description: {freq} heating degree days below {thresh}.

Notes

This index intentionally differs from its ECA&D equivalent: HD17. In HD17, values below zero are not clipped before the sum. The present definition should provide a better representation of the energy demand for heating buildings to the given threshold.

Let TG_{ij} be the daily mean temperature at day i of period j. Then the heating degree days are:

$$HD17_j = \sum_{i=1}^{I} (17 - TG_{ij}) | TG_{ij} < 17)$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.HI(`*tas: Union[DataArray, str] = 'tas', tasmax: Union[DataArray, str] = 'tasmax', lat: Union[DataArray, str] = 'lat', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Huglin Heliothermal Index. (realm: atmos)

Growing-degree days with a base of 10°C and adjusted for latitudes between 40°N and 50°N for April to September (Northern Hemisphere; October to March in Southern Hemisphere). Originally proposed in [Huglin1978]. Used as a heat-summation metric in viticulture agroclimatology.

This indicator will check for missing values according to the method "from_context". Based on indice *huglin_index()*. With injected parameters: thresh=10 degC, method=icclim, start_date=04-01, end_date=11-01.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **lat** (*str or DataArray*) – Latitude coordinate. Default : *ds.lat*. [Required units : []]
- **freq** (*offset alias (string)*) – Resampling frequency (default: "YS"; For Southern Hemisphere, should be "AS-JUL"). Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

HI (*DataArray*) – Huglin heliothermal index (Summation of ((Tmean + Tmax)/2 - {thresh}) * Latitude-based day-length coefficient (*k*), for days between 1 April and 31 October) comment: Metric originally published in Huglin, 1978. Also presented by ECAD/KNMI for ICCLIM, 2013. description: Heat-summation index for agroclimatic suitability estimation, developed specifically for viticulture. Considers daily Tmin and Tmax with a base of {thresh}, typically between 1 April and 30 September. Integrates a day-length coefficient calculation for higher latitudes.

Notes

Let TX_i and TG_i be the daily maximum and mean temperature at day i and T_{thresh} the base threshold needed for heat summation (typically, 10 degC). A day-length multiplication, k, based on latitude, lat, is also considered. Then the Huglin heliothermal index for dates between 1 April and 30 September is:

$$HI = \sum_{i=\text{April 1}}^{\text{September 30}} \left(\frac{TX_i + TG_i)}{2} - T_{thresh} \right) * k$$

For the *smoothed* method, the day-length multiplication factor, k, is calculated as follows:

$$k = f(lat) = \begin{cases} 1, & \text{if } |lat| <= 40 \\ 1 + ((abs(lat) - 40)/10) * 0.06, & \text{if } 40 < |lat| <= 50 \\ NaN, & \text{if } |lat| > 50 \end{cases}$$

For compatibility with ICCLIM, *end_ date* should be set to *11-01*, *method* should be set to *icclim*. The day-length multiplication factor, k, is calculated as follows:

$$k = f(lat) = \begin{cases} 1.0, & \text{if } |lat| <= 40 \\ 1.02, & \text{if } 40 < |lat| <= 42 \\ 1.03, & \text{if } 42 < |lat| <= 44 \\ 1.04, & \text{if } 44 < |lat| <= 46 \\ 1.05, & \text{if } 46 < |lat| <= 48 \\ 1.06, & \text{if } 48 < |lat| <= 50 \\ NaN, & \text{if } |lat| > 50 \end{cases}$$

A more robust day-length calculation based on latitude, calendar, day-of-year, and obliquity is available with *method="jones"*. See: `xclim.indices.generic.day_lengths()` or [Hall&Jones2010]_ for more information.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.ID`(*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of ice/freezing days. (realm: atmos)

Number of days where daily maximum temperatures are below a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *ice_days()*. With injected parameters: thresh=0 degC.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
ID (*DataArray*) – Ice days (TX<0°C) (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with maximum daily temperature below {thresh}.

Notes

Let TX_{ij} be the daily maximum temperature at day i of period j, and :math`TT` the threshold. Then counted is the number of days where:

$$TX_{ij} < TT$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.PRCPTOT`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Accumulated total precipitation (solid and liquid) during wet days (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *prcptot()*. With injected parameters: thresh=1 mm/day.

Parameters

- **pr** (*str or DataArray*) – Total precipitation flux [mm d-1], [mm week-1], [mm month-1] or similar. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
PRCPTOT (*DataArray*) – Precipitation sum over wet days (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum over days description: {freq} total precipitation over wet days, defined as days where precipitation exceeds {thresh}.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.R10mm`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Wet days. (realm: atmos)

Return the total number of days during period with precipitation over threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *wetdays()*. With injected parameters: thresh=10 mm/day.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

R10mm (*DataArray*) – Heavy precipitation days (precipitation10 mm) (number_of_days_with_lwe_thickness_of_precipitation_amount_at_or_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with daily precipitation over {thresh}.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.R20mm`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Wet days. (realm: atmos)

Return the total number of days during period with precipitation over threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *`wetdays()`*. With injected parameters: thresh=20 mm/day.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

R20mm (*DataArray*) – Very heavy precipitation days (precipitation20 mm) (number_of_days_with_lwe_thickness_of_precipitation_amount_at_or_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with daily precipitation over {thresh}.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.R75p`(*pr: Union[DataArray, str] = 'pr'*, *pr_per: Union[DataArray, str] = 'pr_per'*, ***, *freq: str = 'YS'*, *bootstrap: bool = False*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of wet days with daily precipitation over a given percentile. (realm: atmos)

Number of days over period where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *days_over_precip_thresh()*. With injected parameters: thresh=1 mm/day.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – 75th percentile of wet day precipitation flux. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

days_over_precip_thresh (*DataArray*) – Count of days with daily precipitation above the given percentile [days]. (number_of_days_with_lwe_thickness_of_precipitation_amount_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with precipitation above the {pr_per_thresh}th percentile of {pr_per_period} period. Only days with at least {thresh} are counted.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.R75pTOT`(*pr: Union[DataArray, str] = 'pr'*, *pr_per: Union[DataArray, str] = 'pr_per'*, ***, *freq: str = 'YS'*, *bootstrap: bool = False*, *ds: Dataset = None*, ***indexer*) → DataArray

Fraction of precipitation due to wet days with daily precipitation over a given percentile. (realm: atmos)

Percentage of the total precipitation over period occurring in days where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *fraction_over_precip_thresh()*. With injected parameters: thresh=1 mm/day.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – 75th percentile of wet day precipitation flux. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

R75pTOT (*DataArray*) – Precipitation fraction due to moderate wet days (>75th percentile) description: {freq} fraction of total precipitation due to days with precipitation above {pr_per_thresh}th percentile of {pr_per_period} period. Only days with at least {thresh} are included in the total.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.R95p`(*pr: Union[DataArray, str] = 'pr', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*) → DataArray

Number of wet days with daily precipitation over a given percentile. (realm: atmos)

Number of days over period where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *days_over_precip_thresh()*. With injected parameters: thresh=1 mm/day.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – 95th percentile of wet day precipitation flux. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time

series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.

- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

days_over_precip_thresh (*DataArray*) – Count of days with daily precipitation above the given percentile [days]. (number_of_days_with_lwe_thickness_of_precipitation_amount_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with precipitation above the {pr_per_thresh}th percentile of {pr_per_period} period. Only days with at least {thresh} are counted.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.R95pTOT`(*pr: Union[DataArray, str] = 'pr', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*) → DataArray

Fraction of precipitation due to wet days with daily precipitation over a given percentile. (realm: atmos)

Percentage of the total precipitation over period occurring in days where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *`fraction_over_precip_thresh()`*. With injected parameters: thresh=1 mm/day.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – 95th percentile of wet day precipitation flux. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

R95pTOT (*DataArray*) – Precipitation fraction due to very wet days (>95th percentile) description: {freq} fraction of total precipitation due to days with precipitation above {pr_per_thresh}th percentile of {pr_per_period} period. Only days with at least {thresh} are included in the total.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.R99p`(*pr: Union[DataArray, str] = 'pr'*, *pr_per: Union[DataArray, str] = 'pr_per'*, ***, *freq: str = 'YS'*, *bootstrap: bool = False*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of wet days with daily precipitation over a given percentile. (realm: atmos)

Number of days over period where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *days_over_precip_thresh()*. With injected parameters: thresh=1 mm/day.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – 99th percentile of wet day precipitation flux. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

days_over_precip_thresh (*DataArray*) – Count of days with daily precipitation above the given percentile [days]. (number_of_days_with_lwe_thickness_of_precipitation_amount_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with precipitation above the {pr_per_thresh}th percentile of {pr_per_period} period. Only days with at least {thresh} are counted.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.R99pTOT`(*pr: Union[DataArray, str] = 'pr'*, *pr_per: Union[DataArray, str] = 'pr_per'*, ***, *freq: str = 'YS'*, *bootstrap: bool = False*, *ds: Dataset = None*, ***indexer*) → DataArray

Fraction of precipitation due to wet days with daily precipitation over a given percentile. (realm: atmos)

Percentage of the total precipitation over period occurring in days where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *fraction_over_precip_thresh()*. With injected parameters: thresh=1 mm/day.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – 99th percentile of wet day precipitation flux. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

R99pTOT (*DataArray*) – Precipitation fraction due to extremely wet days (>99th percentile) description: {freq} fraction of total precipitation due to days with precipitation above {pr_per_thresh}th percentile of {pr_per_period} period. Only days with at least {thresh} are included in the total.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.RR`(*pr: Union[DataArray, str] = 'pr'*, ***, *thresh: str = '0 degC'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Accumulated total precipitation (solid and liquid) (realm: atmos)

Resample the original daily mean precipitation flux and accumulate over each period. If a daily temperature is provided, the *phase* keyword can be used to sum precipitation of a given phase only. When the temperature is under the provided threshold, precipitation is assumed to be snow, and liquid rain otherwise. This indice is agnostic to the type of daily temperature (tas, tasmax or tasmin) given.

This indicator will check for missing values according to the method "from_context". Based on indice *precip_accumulation()*. With injected parameters: tas=None, phase=None.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold of *tas* over which the precipication is assumed to be liquid rain. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

RR (*DataArray*) – Precipitation sum (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum over days description: {freq} total precipitation

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$

If tas and phase are given, the corresponding phase precipitation is estimated before computing the accumulation, using one of *snowfall_approximation* or *rain_approximation* with the *binary* method.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.RR1`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Wet days. (realm: atmos)

Return the total number of days during period with precipitation over threshold.

This indicator will check for missing values according to the method "from_context". Based on indice `wetdays()`. With injected parameters: thresh=1 mm/day.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

RR1 (*DataArray*) – Wet days (RR1 mm) (number_of_days_with_lwe_thickness_of_precipitation_amount_at_or_above_threshold)

[days] cell_methods: time: sum over days description: {freq} number of days with daily precipitation over {thresh}.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.RX1day`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Highest 1-day precipitation amount for a period (frequency). (realm: atmos)

Resample the original daily total precipitation temperature series by taking the max over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *max_1day_precipitation_amount()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation values. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

RX1day (*DataArray*) – Highest 1-day precipitation amount (lwe_thickness_of_precipitation_amount) [mm/day] cell_methods: time: maximum over days description: {freq} maximum 1-day total precipitation

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j:

$$PRx_{ij} = max(PR_{ij})$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.RX5day`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Highest precipitation amount cumulated over a n-day moving window. (realm: atmos)

Calculate the n-day rolling sum of the original daily total precipitation series and determine the maximum value over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *max_n_day_precipitation_amount()*. With injected parameters: window=5.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation values. Default : *ds.pr*. [Required units : [precipitation]]

- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
RX5day (*DataArray*) – Highest 5-day precipitation amount (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: maximum over days description: {freq} maximum {window}-day total precipitation.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.SD(`*snd: Union[DataArray, str] = 'snd', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Mean of daily average snow depth. (realm: atmos)

Resample the original daily mean snow depth series by taking the mean over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *`snow_depth()`*.

Parameters

- **snd** (*str or DataArray*) – Default : *ds.snd*. [Required units : [length]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
SD (*DataArray*) – Mean of daily snow depth (surface_snow_thickness) [cm] cell_methods: time: mean over days description: {freq} mean of daily mean snow depth.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.SD1(`*snd: Union[DataArray, str] = 'snd', *, freq: str = 'AS-JUL', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with snow depth above a threshold. (realm: atmos)

Number of days where surface snow depth is greater or equal to given threshold. WARNING: The default *freq* is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *`snow_cover_duration()`*. With injected parameters: thresh=1 cm.

Parameters

- **snd** (*str or DataArray*) – Surface snow thickness. Default : *ds.snd*. [Required units : [length]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
SD1 (*DataArray*) – Snow days (SD1 cm) [days] description: {freq} number of days with snow depth greater or equal to {thresh}

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.SD50cm`(*snd: Union[DataArray, str] = 'snd'*, ***, *freq: str = 'AS-JUL'*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of days with snow depth above a threshold. (realm: atmos)

Number of days where surface snow depth is greater or equal to given threshold. WARNING: The default *freq* is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *snow_cover_duration()*. With injected parameters: thresh=50 cm.

Parameters

- **snd** (*str or DataArray*) – Surface snow thickness. Default : *ds.snd*. [Required units : [length]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
SD50cm (*DataArray*) – Snow days (SD50 cm) [days] description: {freq} number of days with snow depth greater or equal to {thresh}

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.SD5cm`(*snd: Union[DataArray, str] = 'snd'*, ***, *freq: str = 'AS-JUL'*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of days with snow depth above a threshold. (realm: atmos)

Number of days where surface snow depth is greater or equal to given threshold. WARNING: The default *freq* is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *snow_cover_duration()*. With injected parameters: thresh=5 cm.

Parameters

- **snd** (*str or DataArray*) – Surface snow thickness. Default : *ds.snd*. [Required units : [length]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.

- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
SD5cm (*DataArray*) – Snow days (SD5 cm) [days] description: {freq} number of days with snow depth greater or equal to {thresh}

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.SDII`(*pr: Union[DataArray, str] = 'pr', *, freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Average daily precipitation intensity. (realm: atmos)

Return the average precipitation over wet days.

This indicator will check for missing values according to the method "from_context". Based on indice *`daily_pr_intensity()`*. With injected parameters: thresh=1 mm/day.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
sdii (*DataArray*) – Average precipitation during wet days (SDII) (lwe_thickness_of_precipitation_amount) [mm/day] description: {freq} Simple Daily Intensity Index (SDII) : {freq} average precipitation for days with daily precipitation over {thresh}. This indicator is also known as the 'Simple Daily Intensity Index' (SDII).

Notes

Let $\mathbf{p} = p_0, p_1, \ldots, p_n$ be the daily precipitation and *thresh* be the precipitation threshold defining wet days. Then the daily precipitation intensity is defined as

$$\frac{\sum_{i=0}^n p_i [p_i \leq thresh]}{\sum_{i=0}^n [p_i \leq thresh]}$$

where $[P]$ is 1 if P is true, and 0 if false.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.SU`(*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of days with tasmax above a threshold (number of summer days). (realm: atmos)

Number of days where daily maximum temperature exceeds a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_days_above()*. With injected parameters: thresh=25 degC.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
SU (*DataArray*) – Summer days (TX>25°C) (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days where daily maximum temperature exceeds {thresh}.

Notes

Let TX_{ij} be the daily maximum temperature at day i of period j. Then counted is the number of days where:

$$TX_{ij} > Threshold[]$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TG`(*tas: Union[DataArray, str] = 'tas'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Mean of daily average temperature. (realm: atmos)

Resample the original daily mean temperature series by taking the mean over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *tg_mean()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
tg_mean (*DataArray*) – Mean daily mean temperature (air_temperature) [K] cell_methods: time: mean over days description: {freq} mean of daily mean temperature.

Notes

Let TN_i be the mean daily temperature of day i, then for a period p starting at day a and finishing on day b:

$$TG_p = \frac{\sum_{i=a}^{b} TN_i}{b - a + 1}$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TG10p`(*tas: Union[DataArray, str] = 'tas', tas_per: Union[DataArray, str] = 'tas_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*) → DataArray

Number of days with daily mean temperature below the 10th percentile. (realm: atmos)

Number of days with daily mean temperature below the 10th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tg10p()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **tas_per** (*str or DataArray*) – 10th percentile of daily mean temperature. Default : *ds.tas_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
TG10p (*DataArray*) – Days with TG<10th percentile of daily mean temperature (cold days) (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with mean daily temperature below

the {tas_per_thresh}th percentile(s). A {tas_per_window} day(s) window, centred on each calendar day in the {tas_per_period} period, is used to compute the {tas_per_thresh}th percentile(s).

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TG90p(`*tas: Union[DataArray, str] = 'tas', tas_per: Union[DataArray, str] = 'tas_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily mean temperature over the 90th percentile. (realm: atmos)

Number of days with daily mean temperature over the 90th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tg90p()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **tas_per** (*str or DataArray*) – 90th percentile of daily mean temperature. Default : *ds.tas_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

TG90p (*DataArray*) – Days with TG>90th percentile of daily mean temperature (warm days) (days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with mean daily temperature above the the {tas_per_thresh}th percentile(s). A {tas_per_window} day(s) window, centred on each calendar day in the {tas_per_period} period, is used to compute the {tas_per_thresh}th percentile(s).

Notes

The 90th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TGn`(*tas: Union[DataArray, str] = 'tas'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Lowest mean temperature. (realm: atmos)

Minimum of daily mean temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *`tg_min()`*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tg_min (*DataArray*) – Minimum daily mean temperature (air_temperature) [K] cell_methods: time: minimum over days description: {freq} minimum of daily mean temperature.

Notes

Let TG_{ij} be the mean temperature at day i of period j. Then the minimum daily mean temperature for period j is:

$$TGn_j = min(TG_{ij})$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TGx`(*tas: Union[DataArray, str] = 'tas'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Highest mean temperature. (realm: atmos)

The maximum of daily mean temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *`tg_max()`*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
tg_max (*DataArray*) – Maximum daily mean temperature (air_temperature) [K] cell_methods: time: maximum over days description: {freq} maximum of daily mean temperature.

Notes

Let TN_{ij} be the mean temperature at day i of period j. Then the maximum daily mean temperature for period j is:

$$TNx_j = max(TN_{ij})$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TN(`*tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Mean minimum temperature. (realm: atmos)

Mean of daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *`tn_mean()`*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
tn_mean (*DataArray*) – Mean daily minimum temperature (air_temperature) [K] cell_methods: time: mean over days description: {freq} mean of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then mean values in period j are given by:

$$TN_{ij} = \frac{\sum_{i=1}^{I} TN_{ij}}{I}$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TN10p(`*tasmin: Union[DataArray, str] = 'tasmin', tasmin_per: Union[DataArray, str] = 'tasmin_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily minimum temperature below the 10th percentile. (realm: atmos)

Number of days with daily minimum temperature below the 10th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tn10p()*.

Parameters

- **tasmin** (*str or DataArray*) – Mean daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmin_per** (*str or DataArray*) – 10th percentile of daily minimum temperature. Default : *ds.tasmin_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

TN10p (*DataArray*) – Days with TN<10th percentile of daily minimum temperature (cold nights) (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with minimum daily temperature below the the {tasmin_per_thresh}th percentile(s). A {tasmin_per_window} day(s) window, centred on each calendar day in the {tasmin_per_period} period, is used to compute the {tasmin_per_thresh}th percentile(s).

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TN90p(`*tasmin: Union[DataArray, str] = 'tasmin', tasmin_per: Union[DataArray, str] = 'tasmin_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily minimum temperature over the 90th percentile. (realm: atmos)

Number of days with daily minimum temperature over the 90th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *`tn90p()`*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmin_per** (*str or DataArray*) – 90th percentile of daily minimum temperature. Default : *ds.tasmin_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

TN90p (*DataArray*) – Days with TN>90th percentile of daily minimum temperature (warm nights) (days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with minimum daily temperature above the the {tasmin_per_thresh}th percentile(s). A {tasmin_per_window} day(s) window, centred on each calendar day in the {tasmin_per_period} period, is used to compute the {tasmin_per_thresh}th percentile(s).

Notes

The 90th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TNn`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Lowest minimum temperature. (realm: atmos)

Minimum of daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_min()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn_min (*DataArray*) – Minimum daily minimum temperature (air_temperature) [K] cell_methods: time: minimum over days description: {freq} minimum of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then the minimum daily minimum temperature for period j is:

$$TNn_j = min(TN_{ij})$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TNx`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Highest minimum temperature. (realm: atmos)

The maximum of daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_max()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn_max (*DataArray*) – Maximum daily minimum temperature (air_temperature) [K] cell_methods: time: maximum over days description: {freq} maximum of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then the maximum daily minimum temperature for period j is:

$$TNx_j = max(TN_{ij})$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TR`(*tasmin: Union[DataArray, str] = 'tasmin'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of days with tasmin above a threshold (number of tropical nights). (realm: atmos)

Number of days where daily minimum temperature exceeds a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_days_above()*. With injected parameters: thresh=20 degC.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

TR (*DataArray*) – Tropical nights (TN>20°C) (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of Tropical Nights : defined as days with minimum daily temperature above {thresh}.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} > Threshold[]$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TX(`*tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Mean max temperature. (realm: atmos)

The mean of daily maximum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice `tx_mean()`.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx_mean (*DataArray*) – Mean daily maximum temperature (air_temperature) [K] cell_methods: time: mean over days description: {freq} mean of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then mean values in period j are given by:

$$TX_{ij} = \frac{\sum_{i=1}^{I} TX_{ij}}{I}$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TX10p(`*tasmax: Union[DataArray, str] = 'tasmax', tasmax_per: Union[DataArray, str] = 'tasmax_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily maximum temperature below the 10th percentile. (realm: atmos)

Number of days with daily maximum temperature below the 10th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice `tx10p()`.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **tasmax_per** (*str or DataArray*) – 10th percentile of daily maximum temperature. Default : *ds.tasmax_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

TX10p (*DataArray*) – Days with TX<10th percentile of daily maximum temperature (cold day-times) (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with maximum daily temperature below the {tasmax_per_thresh}th percentile(s). A {tasmax_per_window} day(s) window, centred on each calendar day in the {tasmax_per_period} period, is used to compute the {tasmax_per_thresh}th percentile(s).

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TX90p(`*tasmax: Union[DataArray, str] = 'tasmax', tasmax_per: Union[DataArray, str] = 'tasmax_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily maximum temperature over the 90th percentile. (realm: atmos)

Number of days with daily maximum temperature over the 90th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice `tx90p()`.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]

- **tasmax_per** (*str or DataArray*) – 90th percentile of daily maximum temperature. Default : *ds.tasmax_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

TX90p (*DataArray*) – Days with TX>90th percentile of daily maximum temperature (warm day-times) (days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with maximum daily temperature above the {tasmax_per_thresh}th percentile(s). A {tasmax_per_window} day(s) window, centred on each calendar day in the {tasmax_per_period} period, is used to compute the {tasmax_per_thresh}th percentile(s).

Notes

The 90th percentile should be computed for a 5-day window centered on each calendar day for a reference period.

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TXn`(*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Lowest max temperature. (realm: atmos)

The minimum of daily maximum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_min()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx_min (*DataArray*) – Minimum daily maximum temperature (air_temperature) [K] cell_methods: time: minimum over days description: {freq} minimum of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then the minimum daily maximum temperature for period j is:

$$TXn_j = min(TX_{ij})$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.TXx`(*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Highest max temperature. (realm: atmos)

The maximum value of daily maximum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_max()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx_max (*DataArray*) – Maximum daily maximum temperature (air_temperature) [K] cell_methods: time: maximum over days description: {freq} maximum of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then the maximum daily maximum temperature for period j is:

$$TXx_j = max(TX_{ij})$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.WD`(*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', tas_per: Union[DataArray, str] = 'tas_per', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

warm and dry days (realm: atmos)

Returns the total number of days where "warm" and "Dry" conditions coincide.

This indicator will check for missing values according to the method "from_context". Based on indice *warm_and_dry_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature values Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **tas_per** (*str or DataArray*) – Daily 75th percentile of temperature. Default : *ds.tas_per*. [Required units : [temperature]]
- **pr_per** (*str or DataArray*) – Daily 25th percentile of wet day precipitation flux. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

WD (*DataArray*) – Warm and dry days [days] cell_methods: time: sum over days description: {freq} number of days where tas > {tas_per_thresh}th percentile and pr < {pr_per_thresh}th percentile

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [warm_dry_days].

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.WSDI(`*tasmax: Union[DataArray, str] = 'tasmax', tasmax_per: Union[DataArray, str] = 'tasmax_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None*`)` → DataArray

Warm spell duration index. (realm: atmos)

Number of days inside spells of a minimum number of consecutive days where the daily maximum temperature is above the 90th percentile. The 90th percentile should be computed for a 5-day moving window, centered on each calendar day in the 1961-1990 period.

This indicator will check for missing values according to the method "from_context". Based on indice *warm_spell_duration_index()*. With injected parameters: window=6.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **tasmax_per** (*str or DataArray*) – percentile(s) of daily maximum temperature. Default : *ds.tasmax_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

WSDI (*DataArray*) – Warm-spell duration index (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with at least {window} consecutive days where the daily maximum temperature is above the {tasmax_per_thresh}th percentile(s). A {tasmax_per_window} day(s) window, centred on each calendar day in the {tasmax_per_period} period, is used to compute the {tasmax_per_thresh}th percentile(s).

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.WW(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', tas_per: Union[DataArray, str] = 'tas_per', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

warm and wet days (realm: atmos)

Returns the total number of days where "warm" and "wet" conditions coincide.

This indicator will check for missing values according to the method "from_context". Based on indice *warm_and_wet_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature values Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **tas_per** (*str or DataArray*) – Daily 75th percentile of temperature. Default : *ds.tas_per*. [Required units : [temperature]]
- **pr_per** (*str or DataArray*) – Daily 75th percentile of wet day precipitation flux. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

WW (*DataArray*) – Warm and wet days [days] cell_methods: time: sum over days description: {freq} number of days where tas > {tas_per_thresh}th percentile and pr > {pr_per_thresh}th percentile

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [warm_wet_days].

References

European Climate Assessment & Dataset, https://www.ecad.eu/

`xclim.indicators.icclim.iter_indicators()`

Iterate over the (name, indicator) pairs in the icclim indicator module.

`xclim.indicators.icclim.vDTR`(*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Mean absolute day-to-day variation in daily temperature range. (realm: atmos)

Mean absolute day-to-day variation in daily temperature range.

This indicator will check for missing values according to the method "from_context". Based on indice *`daily_temperature_range_variability()`*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
vDTR (*DataArray*) – Mean absolute day-to-day difference in DTR (air_temperature) [K] cell_methods: time range within days time: difference over days time: mean over days description: {freq} mean diurnal temparature range variability (defined as the average day-to-day variation in daily temperature range for the given time period)

Notes

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then calculated is the absolute day-to-day differences in period j is:

$$vDTR_j = \frac{\sum_{i=2}^{I} |(TX_{ij} - TN_{ij}) - (TX_{i-1,j} - TN_{i-1,j})|}{I}$$

References

European Climate Assessment & Dataset, https://www.ecad.eu/

ANUCLIM indices

The ANUCLIM (v6.1) software package BIOCLIM sub-module produces a set of bioclimatic parameters derived values of temperature and precipitation. The methods in this module are wrappers around a subset of corresponding methods of *xclim.indices*.

Furthermore, according to the ANUCLIM user-guide ([ANUCLIM]), input values should be at a weekly or monthly frequency. However, the implementation here expands these definitions and can calculate the result with daily input data.

`xclim.indicators.anuclim.P10_MeanTempWarmestQuarter`(*tas: Union[DataArray, str] = 'tas'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

ANUCLIM Mean temperature of warmest/coldest quarter. (realm: atmos)

The warmest (or coldest) quarter of the year is determined, and the mean temperature of this period is calculated. If the input data frequency is daily ("D") or weekly ("W"), quarters are defined as 13-week periods, otherwise as 3 months.

This indicator will check for missing values according to the method "from_context". Based on indice *`tg_mean_warmcold_quarter()`*. With injected parameters: op=warmest.

Parameters

- **tas** (*str or DataArray*) – Mean temperature at daily, weekly, or monthly frequency. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
P10_MeanTempWarmestQuarter (*DataArray*) – (air_temperature) [K] cell_methods: time: mean

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P11_MeanTempColdestQuarter`(*tas: Union[DataArray, str] = 'tas'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

ANUCLIM Mean temperature of warmest/coldest quarter. (realm: atmos)

The warmest (or coldest) quarter of the year is determined, and the mean temperature of this period is calculated. If the input data frequency is daily ("D") or weekly ("W"), quarters are defined as 13-week periods, otherwise as 3 months.

This indicator will check for missing values according to the method "from_context". Based on indice *tg_mean_warmcold_quarter()*. With injected parameters: op=coldest.

Parameters

- **tas** (*str or DataArray*) – Mean temperature at daily, weekly, or monthly frequency. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P11_MeanTempColdestQuarter (*DataArray*) – (air_temperature) [K] cell_methods: time: mean

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P12_AnnualPrecip`(*pr: Union[DataArray, str] = 'pr'*, ***, *thresh: str = '0 mm/d'*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Accumulated total precipitation. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *prcptot()*.

Parameters

- **pr** (*str or DataArray*) – Total precipitation flux [mm d-1], [mm week-1], [mm month-1] or similar. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold over which precipitation starts being cumulated. Default : 0 mm/d. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P12_AnnualPrecip (*DataArray*) – Annual Precipitation (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P13_PrecipWettestPeriod`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

ANUCLIM precipitation of the wettest/driest day, week, or month, depending on the time step. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice `prcptot_wetdry_period()`. With injected parameters: op=wettest.

Parameters

- **pr** (*str or DataArray*) – Total precipitation flux [mm d-1], [mm week-1], [mm month-1] or similar. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P13_PrecipWettestPeriod (*DataArray*) – (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P14_PrecipDriestPeriod`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

ANUCLIM precipitation of the wettest/driest day, week, or month, depending on the time step. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *prcptot_wetdry_period()*. With injected parameters: op=driest.

Parameters

- **pr** (*str or DataArray*) – Total precipitation flux [mm d-1], [mm week-1], [mm month-1] or similar. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
P14_PrecipDriestPeriod (*DataArray*) – (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P15_PrecipSeasonality`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

ANUCLIM Precipitation Seasonality (C of V). (realm: atmos)

The annual precipitation Coefficient of Variation (C of V) expressed in percent. Calculated as the standard deviation of precipitation values for a given year expressed as a percentage of the mean of those values.

This indicator will check for missing values according to the method "from_context". Based on indice *precip_seasonality()*.

Parameters

- **pr** (*str or DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency. Units need to be defined as a rate (e.g. mm d-1, mm week-1). Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P15_PrecipSeasonality (*DataArray*) – cell_methods: time: standard_deviation description: The standard deviation of the precipitation estimates expressed as a percentage of the mean of those estimates.

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

If input units are in mm s-1 (or equivalent) values are converted to mm/day to avoid potentially small denominator values.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P16_PrecipWettestQuarter`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

ANUCLIM Total precipitation of wettest/driest quarter. (realm: atmos)

The wettest (or driest) quarter of the year is determined, and the total precipitation of this period is calculated. If the input data frequency is daily ("D") or weekly ("W") quarters are defined as 13-week periods, otherwise are 3 months.

This indicator will check for missing values according to the method "from_context". Based on indice `prcptot_wetdry_quarter()`. With injected parameters: op=wettest.

Parameters

- **pr** (*str or DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P16_PrecipWettestQuarter (*DataArray*) – (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P17_PrecipDriestQuarter`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

ANUCLIM Total precipitation of wettest/driest quarter. (realm: atmos)

The wettest (or driest) quarter of the year is determined, and the total precipitation of this period is calculated. If the input data frequency is daily ("D") or weekly ("W") quarters are defined as 13-week periods, otherwise are 3 months.

This indicator will check for missing values according to the method "from_context". Based on indice *prcptot_wetdry_quarter()*. With injected parameters: op=driest.

Parameters

- **pr** (*str or DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P17_PrecipDriestQuarter (*DataArray*) – (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P18_PrecipWarmestQuarter`(*pr: Union[DataArray, str] = 'pr'*, *tas: Union[DataArray, str] = 'tas'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

ANUCLIM Total precipitation of warmest/coldest quarter. (realm: atmos)

The warmest (or coldest) quarter of the year is determined, and the total precipitation of this period is calculated. If the input data frequency is daily ("D) or weekly ("W"), quarters are defined as 13-week periods, otherwise are 3 months.

This indicator will check for missing values according to the method "from_context". Based on indice *prcptot_warmcold_quarter()*. With injected parameters: op=warmest.

Parameters

- **pr** (*str or DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean temperature at daily, weekly, or monthly frequency. Default : *ds.tas*. [Required units : [temperature]]

- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
P18_PrecipWarmestQuarter (*DataArray*) – (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P19_PrecipColdestQuarter(`*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

ANUCLIM Total precipitation of warmest/coldest quarter. (realm: atmos)

The warmest (or coldest) quarter of the year is determined, and the total precipitation of this period is calculated. If the input data frequency is daily ("D) or weekly ("W"), quarters are defined as 13-week periods, otherwise are 3 months.

This indicator will check for missing values according to the method "from_context". Based on indice *prcptot_warmcold_quarter()*. With injected parameters: op=coldest.

Parameters

- **pr** (*str or DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean temperature at daily, weekly, or monthly frequency. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
P19_PrecipColdestQuarter (*DataArray*) – (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P1_AnnMeanTemp`(*tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None*) → DataArray

Mean of daily average temperature. (realm: atmos)

Resample the original daily mean temperature series by taking the mean over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *tg_mean()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P1_AnnMeanTemp (*DataArray*) – Annual Mean Temperature (air_temperature) [K] cell_methods: time: mean

Notes

Let TN_i be the mean daily temperature of day i, then for a period p starting at day a and finishing on day b:

$$TG_p = \frac{\sum_{i=a}^{b} TN_i}{b - a + 1}$$

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P2_MeanDiurnalRange`(*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', op: str = 'mean', ds: Dataset = None*) → DataArray

Statistics of daily temperature range. (realm: atmos)

The mean difference between the daily maximum temperature and the daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *daily_temperature_range()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **op** (*{'min', 'mean', 'max', 'std'}*) – Reduce operation. Can either be a DataArray method or a function that can be applied to a DataArray. Default : mean.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P2_MeanDiurnalRange (*DataArray*) – Mean Diurnal Range [K] cell_methods: time: range

Notes

For a default calculation using *op='mean'* :

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the mean diurnal temperature range in period j is:

$$DTR_j = \frac{\sum_{i=1}^{I}(TX_{ij} - TN_{ij})}{I}$$

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P3_Isothermality(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Isothermality. (realm: atmos)

The mean diurnal range divided by the annual temperature range.

This indicator will check for missing values according to the method "from_context". Based on indice `isothermality()`.

Parameters

- **tasmin** (*str or DataArray*) – Average daily minimum temperature at daily, weekly, or monthly frequency. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Average daily maximum temperature at daily, weekly, or monthly frequency. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P3_Isothermality (*DataArray*) – cell_methods: time: range description: The mean diurnal range (P2) divided by the Annual Temperature Range (P7).

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the output with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P4_TempSeasonality(`*tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

ANUCLIM temperature seasonality (coefficient of variation). (realm: atmos)

The annual temperature coefficient of variation expressed in percent. Calculated as the standard deviation of temperature values for a given year expressed as a percentage of the mean of those temperatures.

This indicator will check for missing values according to the method "from_context". Based on indice *temperature_seasonality()*.

Parameters

- **tas** (*str or DataArray*) – Mean temperature at daily, weekly, or monthly frequency. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P4_TempSeasonality (*DataArray*) – cell_methods: time: standard_deviation description: The standard deviation of the mean temperatures expressed as a percentage of the mean of those temperatures. For this calculation, the mean in degrees Kelvin is used. This avoids the possibility of having to divide by zero, but it does mean that the values are usually quite small.

Notes

For this calculation, the mean in degrees Kelvin is used. This avoids the possibility of having to divide by zero, but it does mean that the values are usually quite small.

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P5_MaxTempWarmestPeriod`(*tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None*) → DataArray

Highest max temperature. (realm: atmos)

The maximum value of daily maximum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_max()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P5_MaxTempWarmestPeriod (*DataArray*) – Max Temperature of Warmest Period (air_temperature) [K] description: The highest maximum temperature in all periods of the year. cell_methods: time: maximum

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then the maximum daily maximum temperature for period j is:

$$TXx_j = max(TX_{ij})$$

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P6_MinTempColdestPeriod`(*tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'YS', ds: Dataset = None*) → DataArray

Lowest minimum temperature. (realm: atmos)

Minimum of daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_min()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P6_MinTempColdestPeriod (*DataArray*) – Min Temperature of Coldest Period (air_temperature) [K] description: The lowest minimum temperature in all periods of the year. cell_methods: time: minimum

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then the minimum daily minimum temperature for period j is:

$$TNn_j = min(TN_{ij})$$

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P7_TempAnnualRange(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Calculate the extreme temperature range as the maximum of daily maximum temperature minus the minimum of daily minimum temperature. (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice `extreme_temperature_range()`.

Parameters

- **tasmin** (*str or DataArray*) – Minimum surface temperature. Default : *ds.tasmin*. [Required units : K]
- **tasmax** (*str or DataArray*) – Maximum surface temperature. Default : *ds.tasmax*. [Required units : K]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P7_TempAnnualRange (*DataArray*) – Temperature Annual Range [K] cell_methods: time: range

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P8_MeanTempWettestQuarter(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

ANUCLIM Mean temperature of wettest/driest quarter. (realm: atmos)

The wettest (or driest) quarter of the year is determined, and the mean temperature of this period is calculated. If the input data frequency is daily ("D") or weekly ("W"), quarters are defined as 13-week periods, otherwise are 3 months.

This indicator will check for missing values according to the method "from_context". Based on indice *`tg_mean_wetdry_quarter()`*. With injected parameters: op=wettest.

Parameters

- **tas** (*str or DataArray*) – Mean temperature at daily, weekly, or monthly frequency. Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of [‘A’] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P8_MeanTempWettestQuarter (*DataArray*) – (air_temperature) [K] cell_methods: time: mean

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.P9_MeanTempDriestQuarter(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

ANUCLIM Mean temperature of wettest/driest quarter. (realm: atmos)

The wettest (or driest) quarter of the year is determined, and the mean temperature of this period is calculated. If the input data frequency is daily (“D”) or weekly (“W”), quarters are defined as 13-week periods, otherwise are 3 months.

This indicator will check for missing values according to the method “from_context”. Based on indice *tg_mean_wetdry_quarter()*. With injected parameters: op=driest.

Parameters

- **tas** (*str or DataArray*) – Mean temperature at daily, weekly, or monthly frequency. Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of [‘A’] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

P9_MeanTempDriestQuarter (*DataArray*) – (air_temperature) [K] cell_methods: time: mean

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

References

ANUCLIM https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

`xclim.indicators.anuclim.iter_indicators()`

Iterate over the (name, indicator) pairs in the anuclim indicator module.

14.2 Indices

14.2.1 Indices library

This module contains climate indices functions operating on *xarray.DataArray*. Most of these functions operate on daily time series, but might accept other sampling frequencies as well. All functions perform units checks to make sure that inputs have the expected dimensions (for example have units of temperature, whether it is celsius, kelvin or fahrenheit), and set the *units* attribute of the output DataArray.

The *calendar*, *fwi*, *generic*, *helpers*, *run_length* and *stats* submodules provide helpers to simplify the implementation of the indices.

Note: Indices functions do not perform missing value checks, and usually do not set CF-Convention attributes (long_name, standard_name, description, cell_methods, etc.). These functionalities are provided by `xclim.indicators.Indicator` instances found in the *`xclim.indicators.atmos`*, *`xclim.indicators.land`* and *`xclim.indicators.seaIce`* modules, documented in *Climate indicators*.

`xclim.indices.base_flow_index(`*q: DataArray*, *freq: str = 'YS'*`)` → DataArray

Base flow index.

Return the base flow index, defined as the minimum 7-day average flow divided by the mean flow.

Parameters

- **q** (*xarray.DataArray*) – Rate of river discharge.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Base flow index.

Notes

Let $\mathbf{q} = q_0, q_1, \ldots, q_n$ be the sequence of daily discharge and $\overline{\mathbf{q}}$ the mean flow over the period. The base flow index is given by:

$$\frac{\min(\mathrm{CMA}_7(\mathbf{q}))}{\overline{\mathbf{q}}}$$

where CMA_7 is the seven days moving average of the daily flow:

$$\mathrm{CMA}_7(q_i) = \frac{\sum_{j=i-3}^{i+3} q_j}{7}$$

`xclim.indices.biologically_effective_degree_days(`*tasmin: xarray.DataArray*, *tasmax: xarray.DataArray*, *lat: xarray.DataArray | None = None*, *thresh_tasmin: str = '10 degC'*, *method: str = 'gladstones'*, *low_dtr: str = '10 degC'*, *high_dtr: str = '13 degC'*, *max_daily_degree_days: str = '9 degC'*, *start_date: DayOfYearStr = '04-01'*, *end_date: DayOfYearStr = '11-01'*, *freq: str = 'YS'*`)` → xarray.DataArray

Biologically effective growing degree days.

Growing-degree days with a base of 10°C and an upper limit of 19°C and adjusted for latitudes between 40°N and 50°N for April to October (Northern Hemisphere; October to April in Southern Hemisphere). A temperature range adjustment also promotes small and large swings in daily temperature range. Used as a heat-summation metric in viticulture agroclimatology.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **lat** (*xarray.DataArray, optional*) – Latitude coordinate.
- **thresh_tasmin** (*str*) – The minimum temperature threshold.
- **method** (*{"gladstones", "icclim", "jones"}*) – The formula to use for the calculation. The "gladstones" integrates a daily temperature range and latitude coefficient. End_date should be "11-01". The "icclim" method ignores daily temperature range and latitude coefficient. End date should be "10-01". The "jones" method integrates axial tilt, latitude, and day-of-year on coefficient. End_date should be "11-01".
- **low_dtr** (*str*) – The lower bound for daily temperature range adjustment (default: 10°C).
- **high_dtr** (*str*) – The higher bound for daily temperature range adjustment (default: 13°C).
- **max_daily_degree_days** (*str*) – The maximum amount of biologically effective degrees days that can be summed daily.
- **start_date** (*DayOfYearStr*) – The hemisphere-based start date to consider (north = April, south = October).
- **end_date** (*DayOfYearStr*) – The hemisphere-based start date to consider (north = October, south = April). This date is non-inclusive.
- **freq** (*str*) – Resampling frequency (default: "YS"; For Southern Hemisphere, should be "AS-JUL").

Returns
xarray.DataArray – Biologically effective growing degree days (BEDD).

Warning: Lat coordinate must be provided if method is "gladstones" or "jones".

Notes

The tasmax ceiling of 19°C is assumed to be the max temperature beyond which no further gains from daily temperature occur. Indice originally published in [Gladstones1992].

Let TX_i and TN_i be the daily maximum and minimum temperature at day i, lat the latitude of the point of interest, $degdays_{max}$ the maximum amount of degrees that can be summed per day (typically, 9). Then the sum of daily biologically effective growing degree day (BEDD) units between 1 April and 31 October is:

$$BEDD_i = \sum_{i=\text{April 1}}^{\text{October 31}} min\left(\left(max\left(\frac{TX_i + TN_i)}{2} - 10, 0\right) * k\right) + TR_{adj}, degdays_{max}\right)$$

$$TR_{adj} = f(TX_i, TN_i) = \begin{cases} 0.25(TX_i - TN_i - 13), & \text{if } (TX_i - TN_i) > 13 \\ 0, & \text{if } 10 < (TX_i - TN_i) < 13 \\ 0.25(TX_i - TN_i - 10), & \text{if } (TX_i - TN_i) < 10 \end{cases}$$

$$k = f(lat) = 1 + \left(\frac{|lat|}{50} * 0.06, \text{if } 40 < |lat| < 50, \text{else } 0\right)$$

A second version of the BEDD (*method="icclim"*) does not consider TR_{adj} and k and employs a different end date (30 September) ([ECAD]). The simplified formula is as follows:

$$BEDD_i = \sum_{i=\text{April 1}}^{\text{September 30}} min\left(max\left(\frac{TX_i + TN_i)}{2} - 10, 0\right), degdays_{max}\right)$$

References

`xclim.indices.blowing_snow(`*snd: DataArray, sfcWind: DataArray, snd_thresh: str = '5 cm', sfcWind_thresh: str = '15 km/h', window: int = 3, freq: str = 'AS-JUL'*`)` → DataArray

Days with blowing snow events.

Number of days where both snowfall over the last days and daily wind speeds are above respective thresholds.

Parameters

- **snd** (*xarray.DataArray*) – Surface snow depth.
- **sfcWind** (*xr.DataArray*) – Wind velocity
- **snd_thresh** (*str*) – Threshold on net snowfall accumulation over the last *window* days.
- **sfcWind_thresh** (*str*) – Wind speed threshold.
- **window** (*int*) – Period over which snow is accumulated before comparing against threshold.

- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray – Number of days where snowfall and wind speeds are above respective thresholds.

`xclim.indices.calm_days`(*sfcWind: DataArray*, *thresh: str = '2 m s-1'*, *freq: str = 'MS'*) → DataArray

Calm days.

The number of days with average near-surface wind speed below threshold.

Parameters

- **sfcWind** (*xarray.DataArray*) – Daily windspeed.
- **thresh** (*str*) – Threshold average near-surface wind speed on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days with average near-surface wind speed below threshold.

Notes

Let WS_{ij} be the windspeed at day i of period j. Then counted is the number of days where:

$$WS_{ij} < Threshold[ms - 1]$$

`xclim.indices.clausius_clapeyron_scaled_precipitation`(*delta_tas: DataArray*, *pr_baseline: DataArray*, *cc_scale_factor: float = 1.07*) → DataArray

Scale precipitation according to the Clausius-Clapeyron relation.

Parameters

- **delta_tas** (*xarray.DataArray*) – Difference in temperature between a baseline climatology and another climatology.
- **pr_baseline** (*xarray.DataArray*) – Baseline precipitation to adjust with Clausius-Clapeyron.
- **cc_scale_factor** (*float (default = 1.07)*) – Clausius Clapeyron scale factor.

Returns
DataArray – Baseline precipitation scaled to other climatology using Clausius-Clapeyron relationship.

Notes

The Clausius-Clapeyron equation for water vapor under typical atmospheric conditions states that the saturation water vapor pressure e_s changes approximately exponentially with temperature

$$\frac{\mathrm{d}e_s(T)}{\mathrm{d}T} \approx 1.07 e_s(T)$$

This function assumes that precipitation can be scaled by the same factor.

> **Warning:** Make sure that *delta_tas* is computed over a baseline compatible with *pr_baseline*. So for example, if *delta_tas* is the climatological difference between a baseline and a future period, then *pr_baseline* should be precipitations over a period within the same baseline.

`xclim.indices.cold_and_dry_days(`*tas: DataArray*, *pr: DataArray*, *tas_per: DataArray*, *pr_per: DataArray*, *freq: str = 'YS'*`)` → DataArray

Cold and dry days.

Returns the total number of days where “Cold” and “Dry” conditions coincide.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature values
- **pr** (*xarray.DataArray*) – Daily precipitation.
- **tas_per** (*xarray.DataArray*) – First quartile of daily mean temperature computed by month.
- **pr_per** (*xarray.DataArray*) – First quartile of daily total precipitation computed by month.

> **Warning:** Before computing the percentiles, all the precipitation below 1mm must be filtered out ! Otherwise, the percentiles will include non-wet days.

- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, – The total number of days where cold and dry conditions coincide.

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [cold_dry_days].

References

`xclim.indices.cold_and_wet_days(`*tas: DataArray*, *pr: DataArray*, *tas_per: DataArray*, *pr_per: DataArray*, *freq: str = 'YS'*`)` → DataArray

Cold and wet days.

Returns the total number of days where “cold” and “wet” conditions coincide.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature values
- **pr** (*xarray.DataArray*) – Daily precipitation.
- **tas_per** (*xarray.DataArray*) – First quartile of daily mean temperature computed by month.
- **pr_per** (*xarray.DataArray*) – Third quartile of daily total precipitation computed by month.

- **freq** (*str*) – Resampling frequency.

> **Warning:** Before computing the percentiles, all the precipitation below 1mm must be filtered out! Otherwise, the percentiles will include non-wet days.

Returns
xarray.DataArray, – The total number of days where cold and wet conditions coincide.

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [cold_wet_days].

References

`xclim.indices.cold_spell_days(`*tas: DataArray*, *thresh: str = '-10 degC'*, *window: int = 5*, *freq: str = 'AS-JUL'*`)` → DataArray

Cold spell days.

The number of days that are part of cold spell events, defined as a sequence of consecutive days with mean daily temperature below a threshold in °C.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature below which a cold spell begins.
- **window** (*int*) – Minimum number of days with temperature below threshold to qualify as a cold spell.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Cold spell days.

Notes

Let T_i be the mean daily temperature on day i, the number of cold spell days during period ϕ is given by

$$\sum_{i\in\phi}\prod_{j=i}^{i+5}[T_j < thresh]$$

where $[P]$ is 1 if P is true, and 0 if false.

`xclim.indices.cold_spell_duration_index(`*tasmin: DataArray*, *tasmin_per: DataArray*, *window: int = 6*, *freq: str = 'YS'*, *bootstrap: bool = False*`)` → DataArray

Cold spell duration index.

Number of days with at least *window* consecutive days where the daily minimum temperature is below the *tasmin_per* percentiles.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmin_per** (*xarray.DataArray*) – nth percentile of daily minimum temperature with *dayofyear* coordinate.
- **window** (*int*) – Minimum number of days with temperature below threshold to qualify as a cold spell.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [time] – Count of days with at least six consecutive days where the daily minimum temperature is below the 10th percentile.

Notes

Let TN_i be the minimum daily temperature for the day of the year i and $TN10_i$ the 10th percentile of the minimum daily temperature over the 1961-1990 period for day of the year i, the cold spell duration index over period ϕ is defined as:

$$\sum_{i\in\phi} \prod_{j=i}^{i+6} [TN_j < TN10_j]$$

where $[P]$ is 1 if P is true, and 0 if false.

References

From the Expert Team on Climate Change Detection, Monitoring and Indices (ETCCDMI).

Examples

# Note that this example does not use a proper 1961-1990 reference period. >>> from xclim.core.calendar import percentile_doy >>> from xclim.indices import cold_spell_duration_index

```
>>> tasmin = xr.open_dataset(path_to_tasmin_file).tasmin.isel(lat=0, lon=0)
>>> tn10 = percentile_doy(tasmin, per=10).sel(percentiles=10)
>>> cold_spell_duration_index(tasmin, tn10)
```

`xclim.indices.cold_spell_frequency`(*tas: DataArray*, *thresh: str = '-10 degC'*, *window: int = 5*, *freq: str = 'AS-JUL'*) → DataArray

Cold spell frequency.

The number of cold spell events, defined as a sequence of consecutive days with mean daily temperature below a threshold.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature below which a cold spell begins.
- **window** (*int*) – Minimum number of days with temperature below threshold to qualify as a cold spell.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [dimensionless] – Cold spell frequency.

`xclim.indices.continuous_snow_cover_end(`*snd: DataArray*, *thresh: str = '2 cm'*, *window: int = 14*, *freq: str = 'AS-JUL'*`)` → DataArray

End date of continuous snow cover.

First day after the start of the continuous snow cover when snow depth is below *threshold* for at least *window* consecutive days. WARNING: The default *freq* is valid for the northern hemisphere.

Parameters

- **snd** (*xarray.DataArray*) – Surface snow thickness.
- **thresh** (*str*) – Threshold snow thickness.
- **window** (*int*) – Minimum number of days with snow depth below threshold.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [dimensionless] – First day after the start of the continuous snow cover when the snow depth goes below a threshold for a minimum duration. If there is no such day, return np.nan.

References

Chaumont D., Mailhot A., Diaconescu E.P., Fournier É., Logan T. 2017: Élaboration du portrait bioclimatique futur du Nunavik – Tome II. [Rapport présenté au Ministère de la forêt, de la faune et des parcs], Ouranos.

`xclim.indices.continuous_snow_cover_start(`*snd: DataArray*, *thresh: str = '2 cm'*, *window: int = 14*, *freq: str = 'AS-JUL'*`)` → DataArray

Start date of continuous snow cover.

Day of year when snow depth is above or equal *threshold* for at least *window* consecutive days. WARNING: The default *freq* is valid for the northern hemisphere.

Parameters

- **snd** (*xarray.DataArray*) – Surface snow thickness.
- **thresh** (*str*) – Threshold snow thickness.
- **window** (*int*) – Minimum number of days with snow depth above or equal to threshold.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [dimensionless] – First day of the year when the snow depth is superior to a threshold for a minimum duration. If there is no such day, return np.nan.

References

Chaumont D., Mailhot A., Diaconescu E.P., Fournier É., Logan T. 2017: Élaboration du portrait bioclimatique futur du Nunavik – Tome II. [Rapport présenté au Ministère de la forêt, de la faune et des parcs], Ouranos.

`xclim.indices.cool_night_index(`*tasmin: DataArray*, *lat: DataArray*, *freq: str = 'YS'*`)` → DataArray

Cool Night Index.

Mean minimum temperature for September (northern hemisphere) or March (Southern hemishere). Used in calculating the Géoviticulture Multicriteria Classification System ([Tonietto&Carbonneau2004]_).

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **lat** (*xarray.DataArray, optional*) – Latitude coordinate.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [degC] – Mean of daily minimum temperature for month of interest.

Notes

Given that this indice only examines September and March months, it is possible to send in DataArrays containing only these timesteps. Users should be aware that due to the missing values checks in wrapped Indicators, datasets that are missing several months will be flagged as invalid. This check can be ignored by setting the following context:

Examples

```
>>> with xclim.set_options(
...     check_missing="skip", data_validation="log"
... ):
...     cni = xclim.atmos.cool_night_index(...)
...
```

References

`xclim.indices.cooling_degree_days(`*tas: DataArray*, *thresh: str = '18 degC'*, *freq: str = 'YS'*`)` → DataArray

Cooling degree days.

Sum of degree days above the temperature threshold at which spaces are cooled.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Temperature threshold above which air is cooled.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time][temperature] – Cooling degree days

Notes

Let x_i be the daily mean temperature at day i. Then the cooling degree days above temperature threshold *thresh* over period ϕ is given by:

$$\sum_{i\in\phi}(x_i - thresh[x_i > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false.

`xclim.indices.corn_heat_units(`*tasmin: DataArray*, *tasmax: DataArray*, *thresh_tasmin: str = '4.44 degC'*, *thresh_tasmax: str = '10 degC'*`)` → DataArray

Corn heat units.

Temperature-based index used to estimate the development of corn crops. Formula adapted from [BootsmaTremblay&Filion1999]_.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – The minimum temperature threshold needed for corn growth.
- **thresh_tasmax** (*str*) – The maximum temperature threshold needed for corn growth.

Returns

xarray.DataArray, [dimensionless] – Daily corn heat units.

Notes

Formula used in calculating the Corn Heat Units for the Agroclimatic Atlas of Quebec [Audet&al2012]_.

The thresholds of 4.44°C for minimum temperatures and 10°C for maximum temperatures were selected following the assumption that no growth occurs below these values.

Let TX_i and TN_i be the daily maximum and minimum temperature at day i. Then the daily corn heat unit is:

$$CHU_i = \frac{YX_i + YN_i}{2}$$

with

$$\begin{aligned} YX_i &= 3.33(TX_i - 10) - 0.084(TX_i - 10)^2, &\text{if } TX_i > 10C \\ YN_i &= 1.8(TN_i - 4.44), &\text{if } TN_i > 4.44C \end{aligned}$$

where YX_i and YN_i is 0 when $TX_i \leq 10C$ and $TN_i \leq 4.44C$, respectively.

References

`xclim.indices.daily_freezethaw_cycles(`*tasmin: DataArray, tasmax: DataArray, thresh_tasmin: str = '0 degC', thresh_tasmax: str = '0 degC', freq: str = 'YS'*`)` → DataArray

Number of days with a diurnal freeze-thaw cycle.

The number of days where Tmax > thresh_tasmax and Tmin <= thresh_tasmin.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – The temperature threshold needed to trigger a freeze event.
- **thresh_tasmax** (*str*) – The temperature threshold needed to trigger a thaw event.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Number of days with a diurnal freeze-thaw cycle

Notes

Let TX_i be the maximum temperature at day i and TN_i be the daily minimum temperature at day i. Then the number of freeze thaw cycles during period ϕ is given by :

$$\sum_{i\in\phi}[TX_i > 0][TN_i < 0]$$

where $[P]$ is 1 if P is true, and 0 if false.

Warning: The *daily_freezethaw_cycles* indice is being deprecated in favour of *multiday_temperature_swing* with *thresh_tasmax='0 degC, thresh_tasmin='0 degC', window=1, op='sum'* by default. The indicator reflects this change. This indice will be removed in a future version of xclim.

`xclim.indices.daily_pr_intensity(`*pr: DataArray, thresh: str = '1 mm/day', freq: str = 'YS'*`)` → DataArray

Average daily precipitation intensity.

Return the average precipitation over wet days.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **thresh** (*str*) – Precipitation value over which a day is considered wet.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [precipitation] – The average precipitation over wet days for each period

Notes

Let $\mathbf{p} = p_0, p_1, \ldots, p_n$ be the daily precipitation and *thresh* be the precipitation threshold defining wet days. Then the daily precipitation intensity is defined as

$$\frac{\sum_{i=0}^{n} p_i [p_i \leq thresh]}{\sum_{i=0}^{n} [p_i \leq thresh]}$$

where $[P]$ is 1 if P is true, and 0 if false.

Examples

The following would compute for each grid cell of file *pr.day.nc* the average precipitation fallen over days with precipitation $>= 5$ mm at seasonal frequency, ie DJF, MAM, JJA, SON, DJF, etc.:

```
>>> from xclim.indices import daily_pr_intensity
>>> pr = xr.open_dataset(path_to_pr_file).pr
>>> daily_int = daily_pr_intensity(pr, thresh="5 mm/day", freq="QS-DEC")
```

`xclim.indices.daily_temperature_range(`*tasmin: DataArray, tasmax: DataArray, freq: str = 'YS', op: str = 'mean'*`)` → DataArray

Statistics of daily temperature range.

The mean difference between the daily maximum temperature and the daily minimum temperature.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **freq** (*str*) – Resampling frequency.
- **op** (*{'min', 'max', 'mean', 'std'} or func*) – Reduce operation. Can either be a DataArray method or a function that can be applied to a DataArray.

Returns

xarray.DataArray, [same units as tasmin] – The average variation in daily temperature range for the given time period.

Notes

For a default calculation using *op='mean'* :

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the mean diurnal temperature range in period j is:

$$DTR_j = \frac{\sum_{i=1}^{I}(TX_{ij} - TN_{ij})}{I}$$

`xclim.indices.daily_temperature_range_variability(`*tasmin: DataArray, tasmax: DataArray, freq: str = 'YS'*`)` → DataArray

Mean absolute day-to-day variation in daily temperature range.

Mean absolute day-to-day variation in daily temperature range.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [same units as tasmin] – The average day-to-day variation in daily temperature range for the given time period.

Notes

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then calculated is the absolute day-to-day differences in period j is:

$$vDTR_j = \frac{\sum_{i=2}^{I} |(TX_{ij} - TN_{ij}) - (TX_{i-1,j} - TN_{i-1,j})|}{I}$$

`xclim.indices.days_over_precip_thresh`(*pr: DataArray, pr_per: DataArray, thresh: str = '1 mm/day', freq: str = 'YS', bootstrap: bool = False*) → DataArray

Number of wet days with daily precipitation over a given percentile.

Number of days over period where the precipitation is above a threshold defining wet days and above a given percentile for that day.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **pr_per** (*xarray.DataArray*) – Percentile of wet day precipitation flux. Either computed daily (one value per day of year) or computed over a period (one value per spatial point).
- **thresh** (*str*) – Precipitation value over which a day is considered wet.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns
xarray.DataArray, [time] – Count of days with daily precipitation above the given percentile [days].

Examples

```
>>> from xclim.indices import days_over_precip_thresh
>>> pr = xr.open_dataset(path_to_pr_file).pr
>>> p75 = pr.quantile(0.75, dim="time", keep_attrs=True)
>>> r75p = days_over_precip_thresh(pr, p75)
```

`xclim.indices.days_with_snow`(*prsn: DataArray, low: str = '0 kg m-2 s-1', high: str = '1E6 kg m-2 s-1', freq: str = 'AS-JUL'*) → DataArray

Days with snow.

Return the number of days where snowfall is within low and high thresholds.

Parameters

- **prsn** (*xr.DataArray*) – Solid precipitation flux.
- **low** (*float*) – Minimum threshold solid precipitation flux.
- **high** (*float*) – Maximum threshold solid precipitation flux.
- **freq** (*str*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling.

Returns

xarray.DataArray, [time] – Number of days where snowfall is between low and high thresholds.

References

Matthews, L., Andrey, J., & Picketts, I. (2017). Planning for Winter Road Maintenance in the Context of Climate Change, Weather, Climate, and Society, 9(3), 521-532, https://doi.org/10.1175/WCAS-D-16-0103.1

`xclim.indices.degree_days_exceedance_date`(*tas: DataArray, thresh: str = '0 degC', sum_thresh: str = '25 K days', op: str = '>', after_date: Optional[DayOfYearStr] = None, freq: str = 'YS'*) → DataArray

Degree days exceedance date.

Day of year when the sum of degree days exceeds a threshold. Degree days are computed above or below a given temperature threshold.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base degree days evaluation.
- **sum_thresh** (*str*) – Threshold of the degree days sum.
- **op** (*{">", "gt", "<", "lt", ">=", "ge", "<=", "le"}*) – If equivalent to '>', degree days are computed as *tas - thresh* and if equivalent to '<', they are computed as *thresh - tas*.
- **after_date** (*str, optional*) – Date at which to start the cumulative sum. In "mm-dd" format, defaults to the start of the sampling period.
- **freq** (*str*) – Resampling frequency. If *after_date* is given, *freq* should be annual.

Returns

xarray.DataArray, [dimensionless] – Degree-days exceedance date.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j, T is the reference threshold and ST is the sum threshold. Then, starting at day :math:i_0:, the degree days exceedance date is the first day k such that

$$\begin{cases} ST < \sum_{i=i_0}^{k} \max(TG_{ij} - T, 0) & \text{if } op \text{ is '>'} \\ ST < \sum_{i=i_0}^{k} \max(T - TG_{ij}, 0) & \text{if } op \text{ is '<'} \end{cases}$$

The resulting k is expressed as a day of year.

Cumulated degree days have numerous applications including plant and insect phenology. See https://en.wikipedia.org/wiki/Growing_degree-day for examples.

`xclim.indices.drought_code(`*tas: xr.DataArray, pr: xr.DataArray, lat: xr.DataArray, snd: xr.DataArray | None = None, dc0: xr.DataArray | None = None, season_mask: xr.DataArray | None = None, season_method: str | None = None, overwintering: bool = False, dry_start: str | None = None, initial_start_up: bool = True, **params*`)`

Drought code (FWI component).

The drought code is part of the Canadian Forest Fire Weather Index System. It is a numeric rating of the average moisture content of organic layers.

Parameters

- **tas** (*xr.DataArray*) – Noon temperature.
- **pr** (*xr.DataArray*) – Rain fall in open over previous 24 hours, at noon.
- **lat** (*xr.DataArray*) – Latitude coordinate
- **snd** (*xr.DataArray*) – Noon snow depth.
- **dc0** (*xr.DataArray*) – Initial values of the drought code.
- **season_mask** (*xr.DataArray, optional*) – Boolean mask, True where/when the fire season is active.
- **season_method** (*{None, "WF93", "LA08", "GFWED"}*) – How to compute the start-up and shutdown of the fire season. If "None", no start-ups or shutdowns are computed, similar to the R fwi function. Ignored if *season_mask* is given.
- **overwintering** (*bool*) – Whether to activate DC overwintering or not. If True, either season_method or season_mask must be given.
- **dry_start** (*{None, "CFS", 'GFWED'}*) – Whether to activate the DC and DMC "dry start" mechanism and which method to use. , see `fire_weather_ufunc()`.
- **initial_start_up** (*bool*) – If True (default), grid points where the fire season is active on the first timestep go through a start_up phase for that time step. Otherwise, previous codes must be given as a continuing fire season is assumed for those points.
- **params** – Any other keyword parameters as defined in *xclim.indices.fwi.fire_weather_ufunc* and in `default_params`.

Returns

xr.DataArray, [dimensionless] – Drought code

Notes

See https://cwfis.cfs.nrcan.gc.ca/background/dsm/fwi, the module's doc and doc of `fire_weather_ufunc()` for more information.

References

Updated source code for calculating fire danger indexes in the Canadian Forest Fire Weather Index System, Y. Wang, K.R. Anderson, and R.M. Suddaby, INFORMATION REPORT NOR-X-424, 2015.

`xclim.indices.dry_days(`*pr: DataArray*, *thresh: str = '0.2 mm/d'*, *freq: str = 'YS'*`)` → DataArray

Dry days.

The number of days with daily precipitation below threshold.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Number of days with daily precipitation below threshold.

Notes

Let PR_{ij} be the daily precipitation at day i of period j. Then counted is the number of days where:

$$\sum PR_{ij} < Threshold[mm/day]$$

`xclim.indices.dry_spell_frequency(`*pr: DataArray*, *thresh: str = '1.0 mm'*, *window: int = 3*, *freq: str = 'YS'*, *op: str = 'sum'*`)` → DataArray

Return the number of dry periods of n days and more.

Periods during which the accumulated or maximal daily precipitation amount on a window of n days is under threshold.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **thresh** (*str*) – Precipitation amount under which a period is considered dry. The value against which the threshold is compared depends on *op* .
- **window** (*int*) – Minimum length of the spells.
- **freq** (*str*) – Resampling frequency.
- **op** (*{"sum","max"}*) – Operation to perform on the window. Default is "sum", which checks that the sum of accumulated precipitation over the whole window is less than the threshold. "max" checks that the maximal daily precipitation amount within the window is less than the threshold. This is the same as verifying that each individual day is below the threshold.

Returns

xarray.DataArray – The {freq} number of dry periods of minimum {window} days.

Examples

```
>>> pr = xr.open_dataset(path_to_pr_file).pr
>>> dry_spell_frequency(pr=pr, op="sum")
>>> dry_spell_frequency(pr=pr, op="max")
```

`xclim.indices.dry_spell_total_length(`*pr: DataArray, thresh: str = '1.0 mm', window: int = 3, op: str = 'sum', freq: str = 'YS', **indexer*`)` → DataArray

Total length of dry spells.

Total number of days in dry periods of a minimum length, during which the maximum or accumulated precipitation within a window of the same length is under a threshold.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **thresh** (*str*) – Accumulated precipitation value under which a period is considered dry.
- **window** (*int*) – Number of days when the maximum or accumulated precipitation is under threshold.
- **op** (*{"max", "sum"}*) – Reduce operation.
- **freq** (*str*) – Resampling frequency.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Indexing is done after finding the dry days, but before finding the spells.

Returns

xarray.DataArray – The {freq} total number of days in dry periods of minimum {window} days.

Notes

The algorithm assumes days before and after the timeseries are "wet", meaning that the condition for being considered part of a dry spell is stricter on the edges. For example, with *window=3* and *op='sum'*, the first day of the series is considered part of a dry spell only if the accumulated precipitation within the first 3 days is under the threshold. In comparison, a day in the middle of the series is considered part of a dry spell if any of the three 3-day periods of which it is part are considered dry (so a total of five days are included in the computation, compared to only 3.)

`xclim.indices.effective_growing_degree_days(`*tasmax: DataArray, tasmin: DataArray, *, thresh: str = '5 degC', method: str = 'bootsma', after_date: DayOfYearStr = '07-01', dim: str = 'time', freq: str = 'YS'*`)` → DataArray

Effective growing degree days.

Growing degree days based on a dynamic start and end of the growing season, as defined in [BootsmaGameda&McKenney2005]_.

Parameters

- **tasmax** (*xr.DataArray*) – Daily mean temperature.
- **tasmin** (*xr.DataArray*) – Daily minimum temperature.
- **thresh** (*str*) – The minimum temperature threshold.

- **method** (*{"bootsma", "qian"}*) – The window method used to determine the temperature-based start date. For "bootsma", the start date is defined as 10 days after the average temperature exceeds a threshold (5 degC). For "qian", the start date is based on a weighted 5-day rolling average, based on *qian_weighted_mean_average()*.
- **after_date** (*str*) – Date of the year after which to look for the first frost event. Should have the format '%m-%d'.
- **dim** (*str*) – Time dimension.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

Notes

The effective growing degree days for a given year $EGDD_i$ can be calculated as follows:

The end date is determined as the day preceding the first day with minimum temperature below 0 degC.

References

`xclim.indices.extreme_temperature_range(`*tasmin: DataArray, tasmax: DataArray, freq: str = 'YS'*`)` → DataArray

Extreme intra-period temperature range.

The maximum of max temperature (TXx) minus the minimum of min temperature (TNn) for the given time period.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [same units as tasmin] – Extreme intra-period temperature range for the given time period.

Notes

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the extreme temperature range in period j is:

$$ETR_j = max(TX_{ij}) - min(TN_{ij})$$

`xclim.indices.fire_season(`*tas: xr.DataArray, snd: xr.DataArray | None = None, method: str = 'WF93', freq: str | None = None, temp_start_thresh: str = '12 degC', temp_end_thresh: str = '5 degC', temp_condition_days: int = 3, snow_condition_days: int = 3, snow_thresh: str = '0.01 m'*`)`

Fire season mask.

Binary mask of the active fire season, defined by conditions on consecutive daily temperatures and, optionally, snow depths.

Parameters

- **tas** (*xr.DataArray*) – Daily surface temperature, cffdrs recommends using maximum daily temperature.
- **snd** (*xr.DataArray, optional*) – Snow depth, used with method == 'LA08'.
- **method** (*{"WF93", "LA08", "GFWED"}*) – Which method to use. "LA08" and "GFWED" need the snow depth.
- **freq** (*str, optional*) – If given only the longest fire season for each period defined by this frequency, Every "seasons" are returned if None, including the short shoulder seasons.
- **temp_start_thresh** (*str*) – Minimal temperature needed to start the season.
- **temp_end_thresh** (*str*) – Maximal temperature needed to end the season.
- **temp_condition_days** (*int*) – Number of days with temperature above or below the thresholds to trigger a start or an end of the fire season.
- **snow_condition_days** (*int*) – Parameters for the fire season determination. See `fire_season()`. Temperature is in degC, snow in m. The *snow_thresh* parameters is also used when *dry_start* is set to "GFWED".
- **snow_thresh** (*str*) – Minimal snow depth level to end a fire season, only used with method "LA08".

Returns
fire_season (*xr.DataArray*) – Fire season mask

References

[Wotton&Flannigan1993]_

[Lawson&Armitage2008]_

`xclim.indices.fire_weather_indexes(`*tas: xr.DataArray, pr: xr.DataArray, sfcWind: xr.DataArray, hurs: xr.DataArray, lat: xr.DataArray, snd: xr.DataArray | None = None, ffmc0: xr.DataArray | None = None, dmc0: xr.DataArray | None = None, dc0: xr.DataArray | None = None, season_mask: xr.DataArray | None = None, season_method: str | None = None, overwintering: bool = False, dry_start: str | None = None, initial_start_up: bool = True, **params*`)`

Fire weather indexes.

Computes the 6 fire weather indexes as defined by the Canadian Forest Service: the Drought Code, the Duff-Moisture Code, the Fine Fuel Moisture Code, the Initial Spread Index, the Build Up Index and the Fire Weather Index.

Parameters

- **tas** (*xr.DataArray*) – Noon temperature.
- **pr** (*xr.DataArray*) – Rain fall in open over previous 24 hours, at noon.

- **sfcWind** (*xr.DataArray*) – Noon wind speed.
- **hurs** (*xr.DataArray*) – Noon relative humidity.
- **lat** (*xr.DataArray*) – Latitude coordinate
- **snd** (*xr.DataArray*) – Noon snow depth, only used if *season_method='LA08'* is passed.
- **ffmc0** (*xr.DataArray*) – Initial values of the fine fuel moisture code.
- **dmc0** (*xr.DataArray*) – Initial values of the Duff moisture code.
- **dc0** (*xr.DataArray*) – Initial values of the drought code.
- **season_mask** (*xr.DataArray, optional*) – Boolean mask, True where/when the fire season is active.
- **season_method** (*{None, "WF93", "LA08", "GFWED"}*) – How to compute the start-up and shutdown of the fire season. If "None", no start-ups or shutdowns are computed, similar to the R fwi function. Ignored if *season_mask* is given.
- **overwintering** (*bool*) – Whether to activate DC overwintering or not. If True, either season_method or season_mask must be given.
- **dry_start** (*{None, 'CFS', 'GFWED'}*) – Whether to activate the DC and DMC "dry start" mechanism or not, see `fire_weather_ufunc()`.
- **initial_start_up** (*bool*) – If True (default), gridpoints where the fire season is active on the first timestep go through a start_up phase for that time step. Otherwise, previous codes must be given as a continuing fire season is assumed for those points.
- **params** – Any other keyword parameters as defined in `fire_weather_ufunc()` and in `default_params`.

Returns

- **DC** (*xr.DataArray, [dimensionless]*)
- **DMC** (*xr.DataArray, [dimensionless]*)
- **FFMC** (*xr.DataArray, [dimensionless]*)
- **ISI** (*xr.DataArray, [dimensionless]*)
- **BUI** (*xr.DataArray, [dimensionless]*)
- **FWI** (*xr.DataArray, [dimensionless]*)

Notes

See https://cwfis.cfs.nrcan.gc.ca/background/dsm/fwi, the module's doc and doc of `fire_weather_ufunc()` for more information.

References

Updated source code for calculating fire danger indexes in the Canadian Forest Fire Weather Index System, Y. Wang, K.R. Anderson, and R.M. Suddaby, INFORMATION REPORT NOR-X-424, 2015.

`xclim.indices.first_day_above`(*tasmin: DataArray*, *thresh: str = '0 degC'*, *after_date: DayOfYearStr = '01-01'*, *window: int = 1*, *freq: str = 'YS'*) → DataArray

First day of temperatures superior to a threshold temperature.

Returns first day of period where a temperature is superior to a threshold over a given number of days, limited to a starting calendar date.

WARNING: The default date and freq are valid for the northern hemisphere.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **after_date** (*str*) – Date of the year after which to look for the first event. Should have the format '%m-%d'.
- **window** (*int*) – Minimum number of days with temperature above threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when minimum temperature is superior to a threshold over a given number of days for the first time. If there is no such day, returns np.nan.

`xclim.indices.first_day_below`(*tasmin: DataArray*, *thresh: str = '0 degC'*, *after_date: DayOfYearStr = '07-01'*, *window: int = 1*, *freq: str = 'YS'*) → DataArray

First day of temperatures inferior to a threshold temperature.

Returns first day of period where a temperature is inferior to a threshold over a given number of days, limited to a starting calendar date.

WARNING: The default date and freq are valid for the northern hemisphere.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **after_date** (*str*) – Date of the year after which to look for the first frost event. Should have the format '%m-%d'.
- **window** (*int*) – Minimum number of days with temperature below threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when minimum temperature is inferior to a threshold over a given number of days for the first time. If there is no such day, returns np.nan.

`xclim.indices.first_snowfall`(*prsn: DataArray*, *thresh: str = '0.5 mm/day'*, *freq: str = 'AS-JUL'*) → DataArray

First day with solid precipitation above a threshold.

Returns the first day of a period where the solid precipitation exceeds a threshold. WARNING: The default *freq* is valid for the northern hemisphere.

Parameters

- **prsn** (*xarray.DataArray*) – Solid precipitation flux.
- **thresh** (*str*) – Threshold precipitation flux on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – First day of the year when the solid precipitation is superior to a threshold. If there is no such day, returns np.nan.

References

Climate Projections for the National Capital Region (2020), Volume 1: Results and Interpretation for Key Climate Indices, Report 193600.00, Prepared for Ottawa by CBCL.

`xclim.indices.fraction_over_precip_thresh`(*pr: DataArray*, *pr_per: DataArray*, *thresh: str = '1 mm/day'*, *freq: str = 'YS'*, *bootstrap: bool = False*) → DataArray

Fraction of precipitation due to wet days with daily precipitation over a given percentile.

Percentage of the total precipitation over period occurring in days where the precipitation is above a threshold defining wet days and above a given percentile for that day.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **pr_per** (*xarray.DataArray*) – Percentile of wet day precipitation flux. Either computed daily (one value per day of year) or computed over a period (one value per spatial point).
- **thresh** (*str*) – Precipitation value over which a day is considered wet.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [dimensionless] – Fraction of precipitation over threshold during wet days.

`xclim.indices.freshet_start`(*tas: DataArray*, *thresh: str = '0 degC'*, *window: int = 5*, *freq: str = 'YS'*) → DataArray

First day consistently exceeding threshold temperature.

Returns first day of period where a temperature threshold is exceeded over a given number of days.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **window** (*int*) – Minimum number of days with temperature above threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when temperature exceeds threshold over a given number of days for the first time. If there is no such day, return np.nan.

Notes

Let x_i be the daily mean temperature at day of the year i for values of i going from 1 to 365 or 366. The start date of the freshet is given by the smallest index i for which

$$\prod_{j=i}^{i+w} [x_j > thresh]$$

is true, where w is the number of days the temperature threshold should be exceeded, and $[P]$ is 1 if P is true, and 0 if false.

`xclim.indices.frost_days(`*tasmin: DataArray*, *thresh: str = '0 degC'*, *freq: str = 'YS'*`)` → DataArray

Frost days index.

Number of days where daily minimum temperatures are below a threshold temperature.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Freezing temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Frost days index.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j and :math`TT` the threshold. Then counted is the number of days where:

$$TN_{ij} < TT$$

`xclim.indices.frost_free_season_end(`*tasmin: DataArray*, *thresh: str = '0.0 degC'*, *mid_date: DayOfYearStr = '07-01'*, *window: int = 5*, *freq: str = 'YS'*`)` → DataArray

End of the frost free season.

Day of the year of the start of a sequence of days with minimum temperatures consistently below a threshold, after a period with minimum temperatures consistently above the same threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.

- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **mid_date** (*str*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'.
- **window** (*int*) – Minimum number of days with temperature below threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when minimum temperature is inferior to a threshold over a given number of days for the first time. If there is no such day or if a frost free season is not detected, returns np.nan. If the frost free season does not end within the time period, returns the last day of the period.

`xclim.indices.frost_free_season_length(`*tasmin: xarray.DataArray, window: int = 5, mid_date: DayOfYearStr | None = '07-01', thresh: str = '0.0 degC', freq: str = 'YS'*`)` → xarray.DataArray

Frost free season length.

The number of days between the first occurrence of at least N (def: 5) consecutive days with minimum daily temperature above a threshold (default: 0℃) and the first occurrence of at least N (def 5) consecutive days with minimum daily temperature below the same threshold A mid date can be given to limit the earliest day the end of season can take. WARNING: The default freq and mid_date values are valid for the northern hemisphere.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **window** (*int*) – Minimum number of days with temperature above threshold to mark the beginning and end of frost free season.
- **mid_date** (*str, optional*) – Date the must be included in the season. It is the earliest the end of the season can be. If None, there is no limit.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Frost free season length.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then counted is the number of days between the first occurrence of at least N consecutive days with:

$$TN_{ij} >= 0$$

and the first subsequent occurrence of at least N consecutive days with:

$$TN_{ij} < 0$$

Examples

```
>>> from xclim.indices import frost_season_length
>>> tasmin = xr.open_dataset(path_to_tasmin_file).tasmin
```

# For the Northern Hemisphere: >>> ffsl_nh = frost_free_season_length(tasmin, freq="YS")

# If working in the Southern Hemisphere, one can use: >>> ffsl_sh = frost_free_season_length(tasmin, freq="AS-JUL")

`xclim.indices.frost_free_season_start`(*tasmin: DataArray*, *thresh: str = '0.0 degC'*, *window: int = 5*, *freq: str = 'YS'*) → DataArray

Start of the frost free season.

Day of the year of the start of a sequence of days with minimum temperatures consistently above or equal to a threshold, after a period with minimum temperatures consistently above the same threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **window** (*int*) – Minimum number of days with temperature above threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when minimum temperature is superior to a threshold over a given number of days for the first time. If there is no such day or if a frost free season is not detected, returns np.nan.

Notes

Let x_i be the daily mean temperature at day of the year i for values of i going from 1 to 365 or 366. The start date of the start of growing season is given by the smallest index i for which:

$$\prod_{j=i}^{i+w} [x_j >= thresh]$$

is true, where w is the number of days the temperature threshold should be met or exceeded, and $[P]$ is 1 if P is true, and 0 if false.

`xclim.indices.frost_season_length`(*tasmin: xarray.DataArray*, *window: int = 5*, *mid_date: DayOfYearStr | None = '01-01'*, *thresh: str = '0.0 degC'*, *freq: str = 'AS-JUL'*) → xarray.DataArray

Frost season length.

The number of days between the first occurrence of at least N (def: 5) consecutive days with minimum daily temperature under a threshold (default: 0℃) and the first occurrence of at least N (def 5) consecutive days with minimum daily temperature above the same threshold A mid date can be given to limit the earliest day the end of season can take. WARNING: The default freq and mid_date values are valid for the northern hemisphere.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.

- **window** (*int*) – Minimum number of days with temperature below threshold to mark the beginning and end of frost season.
- **mid_date** (*str, optional*) – Date the must be included in the season. It is the earliest the end of the season can be. If None, there is no limit.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Frost season length.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then counted is the number of days between the first occurrence of at least N consecutive days with:

$$TN_{ij} > 0$$

and the first subsequent occurrence of at least N consecutive days with:

$$TN_{ij} < 0$$

Examples

```
>>> from xclim.indices import frost_season_length
>>> tasmin = xr.open_dataset(path_to_tasmin_file).tasmin
```

# For the Northern Hemisphere: >>> fsl_nh = frost_season_length(tasmin, freq="AS-JUL")

# If working in the Southern Hemisphere, one can use: >>> fsl_sh = frost_season_length(tasmin, freq="YS")

`xclim.indices.growing_degree_days`(*tas: DataArray, thresh: str = '4.0 degC', freq: str = 'YS'*) → DataArray

Growing degree-days over threshold temperature value.

The sum of degree-days over the threshold temperature.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time][temperature] – The sum of growing degree-days above a given threshold.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then the growing degree days are:

$$GD4_j = \sum_{i=1}^{I} (TG_{ij} - 4 | TG_{ij} > 4)$$

`xclim.indices.growing_season_end(`*tas: DataArray*, *thresh: str = '5.0 degC'*, *mid_date: DayOfYearStr = '07-01'*, *window: int = 5*, *freq: str = 'YS'*`)` → DataArray

End of the growing season.

Day of the year of the start of a sequence of days with mean temperatures consistently below a threshold, after a period with mean temperatures consistently above the same threshold.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **mid_date** (*str*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'.
- **window** (*int*) – Minimum number of days with temperature below threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when temperature is inferior to a threshold over a given number of days for the first time. If there is no such day or if a growing season is not detected, returns np.nan. If the growing season does not end within the time period, returns the last day of the period.

`xclim.indices.growing_season_length(`*tas: DataArray*, *thresh: str = '5.0 degC'*, *window: int = 6*, *mid_date: DayOfYearStr = '07-01'*, *freq: str = 'YS'*`)` → DataArray

Growing season length.

The number of days between the first occurrence of at least six consecutive days with mean daily temperature over a threshold (default: 5℃) and the first occurrence of at least six consecutive days with mean daily temperature below the same threshold after a certain date. (Usually July 1st in the northern emisphere and January 1st in the southern hemisphere.)

WARNING: The default calendar values are only valid for the northern hemisphere.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **window** (*int*) – Minimum number of days with temperature above threshold to mark the beginning and end of growing season.
- **mid_date** (*str*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Growing season length.

Notes

Let TG_{ij} be the mean temperature at day i of period j. Then counted is the number of days between the first occurrence of at least 6 consecutive days with:

$$TG_{ij} > 5$$

and the first occurrence after 1 July of at least 6 consecutive days with:

$$TG_{ij} < 5$$

Examples

```
>>> from xclim.indices import growing_season_length
>>> tas = xr.open_dataset(path_to_tas_file).tas
```

# For the Northern Hemisphere: >>> gsl_nh = growing_season_length(tas, mid_date="07-01", freq="AS")

# If working in the Southern Hemisphere, one can use: >>> gsl_sh = growing_season_length(tas, mid_date="01-01", freq="AS-JUL")

`xclim.indices.growing_season_start(`*tas: DataArray, thresh: str = '5.0 degC', window: int = 5, freq: str = 'YS'*`)` → DataArray

Start of the growing season.

Day of the year of the start of a sequence of days with mean temperatures consistently above or equal to a threshold, after a period with mean temperatures consistently above the same threshold.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **window** (*int*) – Minimum number of days with temperature above threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when temperature is superior to a threshold over a given number of days for the first time. If there is no such day or if a growing season is not detected, returns np.nan.

Notes

Let x_i be the daily mean temperature at day of the year i for values of i going from 1 to 365 or 366. The start date of the start of growing season is given by the smallest index i for which:

$$\prod_{j=i}^{i+w} [x_j >= thresh]$$

is true, where w is the number of days the temperature threshold should be met or exceeded, and $[P]$ is 1 if P is true, and 0 if false.

`xclim.indices.heat_index(`*tasmax: DataArray*, *hurs: DataArray*`)` → DataArray

Daily heat index.

Perceived temperature after relative humidity is taken into account ([Blazejczyk2012]). The index is only valid for temperatures above 20°C.

Parameters

- **tasmax** (*xr.DataArray*) – Maximum daily temperature.
- **hurs** (*xr.DataArray*) – Relative humidity.

Returns

xr.DataArray, [time][temperature] – Heat index for days with temperature above 20°C.

References

Notes

While both the humidex and the heat index are calculated using dew point, the humidex uses a dew point of 7 °C (45 °F) as a base, whereas the heat index uses a dew point base of 14 °C (57 °F). Further, the heat index uses heat balance equations which account for many variables other than vapor pressure, which is used exclusively in the humidex calculation.

`xclim.indices.heat_wave_frequency(`*tasmin: DataArray*, *tasmax: DataArray*, *thresh_tasmin: str = '22.0 degC'*, *thresh_tasmax: str = '30 degC'*, *window: int = 3*, *freq: str = 'YS'*`)` → DataArray

Heat wave frequency.

Number of heat waves over a given period. A heat wave is defined as an event where the minimum and maximum daily temperature both exceeds specific thresholds over a minimum number of days.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – The minimum temperature threshold needed to trigger a heatwave event.
- **thresh_tasmax** (*str*) – The maximum temperature threshold needed to trigger a heatwave event.
- **window** (*int*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Number of heatwave at the requested frequency.

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities ([casati2013]).

In Robinson (2001; [robinson2001]), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

`xclim.indices.heat_wave_index(`*tasmax: DataArray*, *thresh: str = '25.0 degC'*, *window: int = 5*, *freq: str = 'YS'*`)` → DataArray

Heat wave index.

Number of days that are part of a heatwave, defined as five or more consecutive days over 25℃.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to designate a heatwave.
- **window** (*int*) – Minimum number of days with temperature above threshold to qualify as a heatwave.
- **freq** (*str*) – Resampling frequency.

Returns

DataArray, [time] – Heat wave index.

`xclim.indices.heat_wave_max_length(`*tasmin: DataArray*, *tasmax: DataArray*, *thresh_tasmin: str = '22.0 degC'*, *thresh_tasmax: str = '30 degC'*, *window: int = 3*, *freq: str = 'YS'*`)` → DataArray

Heat wave max length.

Maximum length of heat waves over a given period. A heat wave is defined as an event where the minimum and maximum daily temperature both exceeds specific thresholds over a minimum number of days.

By definition heat_wave_max_length must be >= window.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – The minimum temperature threshold needed to trigger a heatwave event.
- **thresh_tasmax** (*str*) – The maximum temperature threshold needed to trigger a heatwave event.
- **window** (*int*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Maximum length of heatwave at the requested frequency.

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities ([casati2013]).

In Robinson (2001; [robinson2001]), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

`xclim.indices.heat_wave_total_length(`*tasmin: DataArray*, *tasmax: DataArray*, *thresh_tasmin: str = '22.0 degC'*, *thresh_tasmax: str = '30 degC'*, *window: int = 3*, *freq: str = 'YS'*`)` → DataArray

Heat wave total length.

Total length of heat waves over a given period. A heat wave is defined as an event where the minimum and maximum daily temperature both exceeds specific thresholds over a minimum number of days. This the sum of all days in such events.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – The minimum temperature threshold needed to trigger a heatwave event.
- **thresh_tasmax** (*str*) – The maximum temperature threshold needed to trigger a heatwave event.
- **window** (*int*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Total length of heatwave at the requested frequency.

Notes

See notes and references of *heat_wave_max_length*

`xclim.indices.heating_degree_days(`*tas: DataArray*, *thresh: str = '17.0 degC'*, *freq: str = 'YS'*`)` → DataArray

Heating degree days.

Sum of degree days below the temperature threshold at which spaces are heated.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time][temperature] – Heating degree days index.

Notes

This index intentionally differs from its ECA&D equivalent: HD17. In HD17, values below zero are not clipped before the sum. The present definition should provide a better representation of the energy demand for heating buildings to the given threshold.

Let TG_{ij} be the daily mean temperature at day i of period j. Then the heating degree days are:

$$HD17_j = \sum_{i=1}^{I}(17 - TG_{ij})|TG_{ij} < 17)$$

`xclim.indices.high_precip_low_temp(`*pr: DataArray, tas: DataArray, pr_thresh: str = '0.4 mm/d', tas_thresh: str = '-0.2 degC', freq: str = 'YS'*`)` → DataArray

Number of days with precipitation above threshold and temperature below threshold.

Number of days where precipitation is greater or equal to some threshold, and temperatures are colder than some threshold. This can be used for example to identify days with the potential for freezing rain or icing conditions.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **tas** (*xarray.DataArray*) – Daily mean, minimum or maximum temperature.
- **pr_thresh** (*str*) – Precipitation threshold to exceed.
- **tas_thresh** (*str*) – Temperature threshold not to exceed.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Count of days with high precipitation and low temperatures.

Example

To compute the number of days with intense rainfall while minimum temperatures dip below -0.2C: >>> pr = xr.open_dataset(path_to_pr_file).pr >>> tasmin = xr.open_dataset(path_to_tasmin_file).tasmin >>> high_precip_low_temp(… pr, tas=tasmin, pr_thresh="10 mm/d", tas_thresh="-0.2 degC" …)

`xclim.indices.hot_spell_frequency(`*tasmax: DataArray, thresh_tasmax: str = '30 degC', window: int = 3, freq: str = 'YS'*`)` → DataArray

Hot spell frequency.

Number of hot spells over a given period. A hot spell is defined as an event where the maximum daily temperature exceeds a specific threshold over a minimum number of days.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmax** (*str*) – The maximum temperature threshold needed to trigger a heatwave event.

- **window** (*int*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [dimensionless] – Number of heatwave at the wanted frequency

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities (Casati et al., 2013).

In Robinson (2001), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

Casati, B., A. Yagouti, and D. Chaumont, 2013: Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. J. Appl. Meteor. Climatol., 52, 2669–2698, https://doi.org/10.1175/JAMC-D-12-0341.1

Robinson, P.J., 2001: On the Definition of a Heat Wave. J. Appl. Meteor., 40, 762–775, https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

`xclim.indices.hot_spell_max_length(`*tasmax: DataArray*, *thresh_tasmax: str = '30 degC'*, *window: int = 1*, *freq: str = 'YS'*`)` → DataArray

Longest hot spell.

Longest spell of high temperatures over a given period.

The longest series of consecutive days with tasmax 30 °C. Here, there is no minimum threshold for number of days in a row that must be reached or exceeded to count as a spell. A year with zero +30 °C days will return a longest spell value of zero.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmax** (*str*) – The maximum temperature threshold needed to trigger a heatwave event.
- **window** (*int*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Maximum length of continuous hot days at the wanted frequency.

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities (Casati et al., 2013).

In Robinson (2001), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

Casati, B., A. Yagouti, and D. Chaumont, 2013: Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. J. Appl. Meteor. Climatol., 52, 2669–2698, https://doi.org/10.1175/JAMC-D-12-0341.1

Robinson, P.J., 2001: On the Definition of a Heat Wave. J. Appl. Meteor., 40, 762–775, https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

`xclim.indices.huglin_index(`*tas: DataArray*, *tasmax: DataArray*, *lat: DataArray*, *thresh: str = '10 degC'*, *method: str = 'smoothed'*, *start_date: DayOfYearStr = '04-01'*, *end_date: DayOfYearStr = '10-01'*, *freq: str = 'YS'*`)` → DataArray

Huglin Heliothermal Index.

Growing-degree days with a base of 10°C and adjusted for latitudes between 40°N and 50°N for April to September (Northern Hemisphere; October to March in Southern Hemisphere). Originally proposed in [Huglin1978]. Used as a heat-summation metric in viticulture agroclimatology.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **lat** (*xarray.DataArray*) – Latitude coordinate.
- **thresh** (*str*) – The temperature threshold.
- **method** (*{"smoothed", "icclim", "jones"}*) – The formula to use for the latitude coefficient calculation.
- **start_date** (*DayOfYearStr*) – The hemisphere-based start date to consider (north = April, south = October).
- **end_date** (*DayOfYearStr*) – The hemisphere-based start date to consider (north = October, south = April). This date is non-inclusive.
- **freq** (*str*) – Resampling frequency (default: "YS"; For Southern Hemisphere, should be "AS-JUL").

Returns

xarray.DataArray, [unitless] – Huglin heliothermal index (HI).

Notes

Let TX_i and TG_i be the daily maximum and mean temperature at day i and T_{thresh} the base threshold needed for heat summation (typically, 10 degC). A day-length multiplication, k, based on latitude, lat, is also considered. Then the Huglin heliothermal index for dates between 1 April and 30 September is:

$$HI = \sum_{i=\text{April 1}}^{\text{September 30}} \left(\frac{TX_i + TG_i)}{2} - T_{thresh} \right) * k$$

For the *smoothed* method, the day-length multiplication factor, k, is calculated as follows:

$$k = f(lat) = \begin{cases} 1, & \text{if } |lat| <= 40 \\ 1 + ((abs(lat) - 40)/10) * 0.06, & \text{if } 40 < |lat| <= 50 \\ NaN, & \text{if } |lat| > 50 \end{cases}$$

For compatibility with ICCLIM, *end_ date* should be set to *11-01*, *method* should be set to *icclim*. The day-length multiplication factor, k, is calculated as follows:

$$k = f(lat) = \begin{cases} 1.0, & \text{if } |lat| <= 40 \\ 1.02, & \text{if } 40 < |lat| <= 42 \\ 1.03, & \text{if } 42 < |lat| <= 44 \\ 1.04, & \text{if } 44 < |lat| <= 46 \\ 1.05, & \text{if } 46 < |lat| <= 48 \\ 1.06, & \text{if } 48 < |lat| <= 50 \\ NaN, & \text{if } |lat| > 50 \end{cases}$$

A more robust day-length calculation based on latitude, calendar, day-of-year, and obliquity is available with *method="jones"*. See: `xclim.indices.generic.day_lengths()` or [Hall&Jones2010]_ for more information.

References

`xclim.indices.humidex`(*tas: xr.DataArray*, *tdps: xr.DataArray | None = None*, *hurs: xr.DataArray | None = None*) → xr.DataArray

Humidex index.

The humidex indicates how hot the air feels to an average person, accounting for the effect of humidity. It can be loosely interpreted as the equivalent perceived temperature when the air is dry.

Parameters

- **tas** (*xarray.DataArray*) – Air temperature.
- **tdps** (*xarray.DataArray,*) – Dewpoint temperature.
- **hurs** (*xarray.DataArray*) – Relative humidity.

Returns

xarray.DataArray, [temperature] – The humidex index.

Notes

The humidex is usually computed using hourly observations of dry bulb and dewpoint temperatures. It is computed using the formula based on [masterton79]:

$$T + \frac{5}{9}\left[e - 10\right]$$

where T is the dry bulb air temperature (°C). The term e can be computed from the dewpoint temperature $T_{dewpoint}$ in °K:

$$e = 6.112 \times \exp(5417.7530\left({\frac{1}{273.16}-\frac{1}{T_{\text{dewpoint}}}}\right)$$

where the constant 5417.753 reflects the molecular weight of water, latent heat of vaporization, and the universal gas constant ([mekis15]). Alternatively, the term e can also be computed from the relative humidity h expressed in percent using [sirangelo20]:

$$e = \frac{h}{100} \times 6.112 * 10^{7.5 T/(T + 237.7)}.$$

The humidex *comfort scale* ([eccc]) can be interpreted as follows:

- 20 to 29 : no discomfort;
- 30 to 39 : some discomfort;
- 40 to 45 : great discomfort, avoid exertion;
- 46 and over : dangerous, possible heat stroke;

Please note that while both the humidex and the heat index are calculated using dew point, the humidex uses a dew point of 7 °C (45 °F) as a base, whereas the heat index uses a dew point base of 14 °C (57 °F). Further, the heat index uses heat balance equations which account for many variables other than vapor pressure, which is used exclusively in the humidex calculation.

References

`xclim.indices.ice_days`(*tasmax: DataArray*, *thresh: str = '0 degC'*, *freq: str = 'YS'*) → DataArray

Number of ice/freezing days.

Number of days where daily maximum temperatures are below a threshold.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh** (*str*) – Freezing temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Number of ice/freezing days.

Notes

Let TX_{ij} be the daily maximum temperature at day i of period j, and :math`TT` the threshold. Then counted is the number of days where:

$$TX_{ij} < TT$$

`xclim.indices.isothermality(`*tasmin: DataArray*, *tasmax: DataArray*, *freq: str = 'YS'*`)` → DataArray

Isothermality.

The mean diurnal range divided by the annual temperature range.

Parameters

- **tasmin** (*xarray.DataArray*) – Average daily minimum temperature at daily, weekly, or monthly frequency.
- **tasmax** (*xarray.DataArray*) – Average daily maximum temperature at daily, weekly, or monthly frequency.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [%] – Isothermality

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the output with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices.jetstream_metric_woollings(`*ua: xarray.DataArray*`)`

Strength and latitude of jetstream.

Identify latitude and strength of maximum smoothed zonal wind speed in the region from 15 to 75°N and -60 to 0°E, using the formula outlined in ([Woollings2010]).

Warning: This metric expects eastward wind component (u) to be on a regular grid (i.e. Plate Carree, 1D lat and lon)

Parameters

ua (*xarray.DataArray*) – Eastward wind component (u) at between 750 and 950 hPa.

Returns

(xarray.DataArray, xarray.DataArray) – Daily time series of latitude of jetstream and Daily time series of strength of jetstream.

References

`xclim.indices.last_snowfall(`*prsn: DataArray*, *thresh: str = '0.5 mm/day'*, *freq: str = 'AS-JUL'*`)` → DataArray

Last day with solid precipitation above a threshold.

Returns the last day of a period where the solid precipitation exceeds a threshold. WARNING: The default freq is valid for the northern hemisphere.

Parameters

- **prsn** (*xarray.DataArray*) – Solid precipitation flux.
- **thresh** (*str*) – Threshold precipitation flux on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Last day of the year when the solid precipitation is superior to a threshold. If there is no such day, returns np.nan.

References

Climate Projections for the National Capital Region (2020), Volume 1: Results and Interpretation for Key Climate Indices, Report 193600.00, Prepared for Ottawa by CBCL.

`xclim.indices.last_spring_frost(`*tas: DataArray*, *thresh: str = '0 degC'*, *before_date: DayOfYearStr = '07-01'*, *window: int = 1*, *freq: str = 'YS'*`)` → DataArray

Last day of temperatures inferior to a threshold temperature.

Returns last day of period where a temperature is inferior to a threshold over a given number of days and limited to a final calendar date.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **before_date** (*str,*) – Date of the year before which to look for the final frost event. Should have the format '%m-%d'.
- **window** (*int*) – Minimum number of days with temperature below threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when temperature is inferior to a threshold over a given number of days for the first time. If there is no such day, returns np.nan.

`xclim.indices.latitude_temperature_index(`*tas: DataArray*, *lat: DataArray*, *lat_factor: float = 75*, *freq: str = 'YS'*`)` → DataArray

Latitude-Temperature Index.

Mean temperature of the warmest month with a latitude-based scaling factor ([Jackson&Cherry1988]_). Used for categorizing wine-growing regions.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.

- **lat** (*xarray.DataArray*) – Latitude coordinate.
- **lat_factor** (*float*) – Latitude factor. Maximum poleward latitude. Default: 75.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [unitless] – Latitude Temperature Index.

Notes

The latitude factor of *75* is provided for examining the poleward expansion of wine-growing climates under scenarios of climate change (modified from [Kenny&Shao1992]_). For comparing 20th century/observed historical records, the original scale factor of *60* is more appropriate.

Let Tn_j be the average temperature for a given month j, lat_f be the latitude factor, and lat be the latitude of the area of interest. Then the Latitude-Temperature Index (LTI) is:

$$LTI = max(TN_j : j = 1..12)(lat_f - |lat|)$$

References

`xclim.indices.liquid_precip_ratio(`*pr: xarray.DataArray, prsn: xarray.DataArray | None = None, tas: xarray.DataArray | None = None, thresh: str = '0 degC', freq: str = 'QS-DEC'*`)` → xarray.DataArray

Ratio of rainfall to total precipitation.

The ratio of total liquid precipitation over the total precipitation. If solid precipitation is not provided, it is approximated with pr, tas and thresh, using the *snowfall_approximation* function with method 'binary'.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **prsn** (*xarray.DataArray, optional*) – Mean daily solid precipitation flux.
- **tas** (*xarray.DataArray, optional*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature under which precipitation is assumed to be solid.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [dimensionless] – Ratio of rainfall to total precipitation.

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$

$$PRwet_{ij}$$

See also:

`winter_rain_ratio`

`xclim.indices.max_1day_precipitation_amount`(*pr: DataArray*, *freq: str = 'YS'*) → DataArray

Highest 1-day precipitation amount for a period (frequency).

Resample the original daily total precipitation temperature series by taking the max over each period.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation values.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as pr] – The highest 1-period precipitation flux value at the given time frequency.

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j:

$$PRx_{ij} = max(PR_{ij})$$

Examples

```
>>> from xclim.indices import max_1day_precipitation_amount
```

# The following would compute for each grid cell the highest 1-day total # at an annual frequency: >>> pr = xr.open_dataset(path_to_pr_file).pr >>> rx1day = max_1day_precipitation_amount(pr, freq="YS")

`xclim.indices.max_n_day_precipitation_amount`(*pr: DataArray*, *window: int = 1*, *freq: str = 'YS'*) → DataArray

Highest precipitation amount cumulated over a n-day moving window.

Calculate the n-day rolling sum of the original daily total precipitation series and determine the maximum value over each period.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation values.
- **window** (*int*) – Window size in days.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [length] – The highest cumulated n-period precipitation value at the given time frequency.

Examples

```
>>> from xclim.indices import max_n_day_precipitation_amount
```

# The following would compute for each grid cell the highest 5-day total precipitation #at an annual frequency: >>> pr = xr.open_dataset(path_to_pr_file).pr >>> out = max_n_day_precipitation_amount(pr, window=5, freq="YS")

`xclim.indices.max_pr_intensity(`*pr: DataArray*, *window: int = 1*, *freq: str = 'YS'*`)` → DataArray

Highest precipitation intensity over a n-hour moving window.

Calculate the n-hour rolling average of the original hourly total precipitation series and determine the maximum value over each period.

Parameters

- **pr** (*xarray.DataArray*) – Hourly precipitation values.
- **window** (*int*) – Window size in hours.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as pr] – The highest cumulated n-hour precipitation intensity at the given time frequency.

Examples

```
>>> from xclim.indices import max_pr_intensity
```

# The following would compute the maximum 6-hour precipitation intensity. # at an annual frequency: # TODO

`xclim.indices.maximum_consecutive_dry_days(`*pr: DataArray*, *thresh: str = '1 mm/day'*, *freq: str = 'YS'*`)` → DataArray

Maximum number of consecutive dry days.

Return the maximum number of consecutive days within the period where precipitation is below a certain threshold.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **thresh** (*str*) – Threshold precipitation on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – The maximum number of consecutive dry days (precipitation < threshold per period).

Notes

Let $\mathbf{p} = p_0, p_1, \ldots, p_n$ be a daily precipitation series and $thresh$ the threshold under which a day is considered dry. Then let $\mathbf{s}$ be the sorted vector of indices i where $[p_i < thresh] \neq [p_{i+1} < thresh]$, that is, the days when the precipitation crosses the threshold. Then the maximum number of consecutive dry days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[p_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indices.maximum_consecutive_frost_days(`*tasmin: DataArray*, *thresh: str = '0.0 degC'*, *freq: str = 'AS-JUL'*`)` → DataArray

Maximum number of consecutive frost days (Tn < 0℃).

The maximum number of consecutive days within the period where the temperature is under a certain threshold (default: 0°C). WARNING: The default freq value is valid for the northern hemisphere.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – The maximum number of consecutive frost days (tasmin < threshold per period).

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily minimum temperature series and $thresh$ the threshold below which a day is considered a frost day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i < thresh] \neq [t_{i+1} < thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive frost free days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indices.maximum_consecutive_frost_free_days(`*tasmin: DataArray*, *thresh: str = '0 degC'*, *freq: str = 'YS'*`)` → DataArray

Maximum number of consecutive frost free days (Tn >= 0℃).

Return the maximum number of consecutive days within the period where the minimum temperature is above or equal to a certain threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – The maximum number of consecutive frost free days (tasmin >= threshold per period).

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily minimum temperature series and $thresh$ the threshold above or equal to which a day is considered a frost free day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i <= thresh] \neq [t_{i+1} <= thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive frost free days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} >= thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indices.maximum_consecutive_tx_days`(*tasmax: DataArray*, *thresh: str = '25 degC'*, *freq: str = 'YS'*) → DataArray

Maximum number of consecutive days with tasmax above a threshold (summer days).

Return the maximum number of consecutive days within the period where the maximum temperature is above a certain threshold.

Parameters

- **tasmax** (*xarray.DataArray*) – Max daily temperature.
- **thresh** (*str*) – Threshold temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – The maximum number of days with tasmax > thresh per periods (summer days).

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily maximum temperature series and $thresh$ the threshold above which a day is considered a summer day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i < thresh] \neq [t_{i+1} < thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive dry days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indices.maximum_consecutive_wet_days`(*pr: DataArray*, *thresh: str = '1 mm/day'*, *freq: str = 'YS'*) → DataArray

Consecutive wet days.

Returns the maximum number of consecutive wet days.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **thresh** (*str*) – Threshold precipitation on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – The maximum number of consecutive wet days.

Notes

Let $\mathbf{x} = x_0, x_1, \ldots, x_n$ be a daily precipitation series and $\mathbf{s}$ be the sorted vector of indices i where $[p_i > thresh] \neq [p_{i+1} > thresh]$, that is, the days when the precipitation crosses the *wet day* threshold. Then the maximum number of consecutive wet days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[x_{s_j} > 0^\circ C]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indices.mean_radiant_temperature`(*rsds: DataArray*, *rsus: DataArray*, *rlds: DataArray*, *rlus: DataArray*, *stat: str = 'average'*) → DataArray

Mean radiant temperature.

The mean radiant temperature is the incidence of radiation on the body from all directions. WARNING: There are some issues in the calculation of mrt in polar regions.

Parameters

- **rsds** (*xr.DataArray*) – Surface Downwelling Shortwave Radiation
- **rsus** (*xr.DataArray*) – Surface Upwelling Shortwave Radiation
- **rlds** (*xr.DataArray*) – Surface Downwelling Longwave Radiation
- **rlus** (*xr.DataArray*) – Surface Upwelling Longwave Radiation
- **stat** (*{'average', 'instant', 'sunlit'}*) – Which statistic to apply. If "average", the average of the cosine of the solar zenith angle is calculated. If "instant", the instantaneous cosine of the solar zenith angle is calculated. If "sunlit", the cosine of the solar zenith angle is calculated during the sunlit period of each interval. If "instant", the instantaneous cosine of the solar zenith angle is calculated. This is necessary if mrt is not None.

Returns

xarray.DataArray, [K] – Mean radiant temperature

Notes

This code was inspired by the *thermofeel* package.

References

Di Napoli, C., Hogan, R.J. & Pappenberger, F. Mean radiant temperature from global-scale numerical weather prediction models. Int J Biometeorol 64, 1233–1245 (2020). https://doi.org/10.1007/s00484-020-01900-5 Brimicombe , C., Di Napoli, C., Quintino, T., Pappenberger, F., Cornforth, R. and Cloke, H., 2021 thermofeel: a python thermal comfort indices library, https://doi.org/10.21957/mp6v-fd16

`xclim.indices.melt_and_precip_max`(*snw: DataArray*, *pr: DataArray*, *window: int = 3*, *freq: str = 'AS-JUL'*) → DataArray

Maximum snow melt and precipitation.

The maximum snow melt plus precipitation over a given number of days expressed in snow water equivalent.

Parameters

- **snw** (*xarray.DataArray*) – Snow amount (mass per area).
- **pr** (*xarray.DataArray*) – Daily precipitation flux.
- **window** (*int*) – Number of days during which the water input is accumulated.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray – The maximum snow melt plus precipitation over a given number of days for each period. [mass/area].

`xclim.indices.multiday_temperature_swing(`*tasmin: DataArray, tasmax: DataArray, thresh_tasmin: str = '0 degC', thresh_tasmax: str = '0 degC', window: int = 1, op: str = 'mean', freq: str = 'YS'*`)` → DataArray

Statistics of consecutive diurnal temperature swing events.

A diurnal swing of max and min temperature event is when Tmax > thresh_tasmax and Tmin <= thresh_tasmin. This indice finds all days that constitute these events and computes statistics over the length and frequency of these events.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – The temperature threshold needed to trigger a freeze event.
- **thresh_tasmax** (*str*) – The temperature threshold needed to trigger a thaw event.
- **window** (*int*) – The minimal length of spells to be included in the statistics.
- **op** (*{'mean', 'sum', 'max', 'min', 'std', 'count'}*) – The statistical operation to use when reducing the list of spell lengths.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – {freq} {op} length of diurnal temperature cycles exceeding thresholds.

Notes

Let TX_i be the maximum temperature at day i and TN_i be the daily minimum temperature at day i. Then freeze thaw spells during a given period are consecutive days where:

$$TX_i > 0 \wedge TN_i < 0$$

This indice returns a given statistic of the found lengths, optionally dropping those shorter than the *window* argument. For example, *window=1* and *op='sum'* returns the same value as `daily_freezethaw_cycles()`.

`xclim.indices.potential_evapotranspiration(`*tasmin: xr.DataArray | None = None, tasmax: xr.DataArray | None = None, tas: xr.DataArray | None = None, lat: xr.DataArray | None = None, method: str = 'BR65', peta: float | None = 0.00516409319477, petb: float | None = 0.0874972822289*`)` → xr.DataArray

Potential evapotranspiration.

The potential for water evaporation from soil and transpiration by plants if the water supply is sufficient, according to a given method.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **lat** (*xarray.DataArray, optional*) – Latitude. If not given, it is sought on tasmin or tas with cf-xarray.
- **method** (*{"baierrobertson65", "BR65", "hargreaves85", "HG85", "thornthwaite48", "TW48", "mcguinnessbordne05", "MB05"}*) – Which method to use, see notes.
- **peta** (*float*) – Used only with method MB05 as a for calculation of PET, see Notes section. Default value resulted from calibration of PET over the UK.
- **petb** (*float*) – Used only with method MB05 as b for calculation of PET, see Notes section. Default value resulted from calibration of PET over the UK.

Returns

xarray.DataArray

Notes

Available methods are:

- "baierrobertson65" or "BR65", based on [BaierRobertson1965]. Requires tasmin and tasmax, daily [D] freq.
- "hargreaves85" or "HG85", based on [Hargreaves1985]. Requires tasmin and tasmax, daily [D] freq. (optional: tas can be given in addition of tasmin and tasmax).
- "mcguinnessbordne05" or "MB05", based on [Tanguy2018]. Requires tas, daily [D] freq, with latitudes 'lat'.
- "thornthwaite48" or "TW48", based on [Thornthwaite1948]. Requires tasmin and tasmax, monthly [MS] or daily [D] freq. (optional: tas can be given instead of tasmin and tasmax).

The McGuinness-Bordne [McGuinness1972] equation is:

$$PET[mmday^{-1}] = a * \frac{S_0}{\lambda}T_a + b * S_0\lambda$$

where a and b are empirical parameters; S_0 is the extraterrestrial radiation [MJ m-2 day-1], assuming a solar constant of 1367 W m-2;
lambda is the latent heat of vaporisation [MJ kg-1] and T_a is the air temperature [°C]. The equation was originally derived for the USA, with $a = 0.0147$ and $b = 0.07353$. The default parameters used here are calibrated for the UK, using the method described in [Tanguy2018].

Methods "BR65", "HG85" and "MB05" use an approximation of the extraterrestrial radiation. See `extraterrestrial_solar_radiation()`.

References

`xclim.indices.prcptot`(*pr: DataArray*, *thresh: str = '0 mm/d'*, *freq: str = 'YS'*) → DataArray

Accumulated total precipitation.

Parameters

- **pr** (*xarray.DataArray*) – Total precipitation flux [mm d-1], [mm week-1], [mm month-1] or similar.
- **thresh** (*str*) – Threshold over which precipitation starts being cumulated.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [length] – Total {freq} precipitation.

`xclim.indices.prcptot_warmcold_quarter`(*pr: DataArray*, *tas: DataArray*, *op: Optional[str] = None*, *freq: str = 'YS'*) → DataArray

ANUCLIM Total precipitation of warmest/coldest quarter.

The warmest (or coldest) quarter of the year is determined, and the total precipitation of this period is calculated. If the input data frequency is daily ("D) or weekly ("W"), quarters are defined as 13-week periods, otherwise are 3 months.

Parameters

- **pr** (*xarray.DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency.
- **tas** (*xarray.DataArray*) – Mean temperature at daily, weekly, or monthly frequency.
- **op** (*{'warmest', 'coldest'}*) – Operation to perform: 'warmest' calculate for the warmest quarter ; 'coldest' calculate for the coldest quarter.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray (*[mm]*) – Precipitation of {op} quarter

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices.prcptot_wetdry_period`(*pr: DataArray*, ***, *op: str*, *freq: str = 'YS'*) → DataArray

ANUCLIM precipitation of the wettest/driest day, week, or month, depending on the time step.

Parameters

- **pr** (*xarray.DataArray*) – Total precipitation flux [mm d-1], [mm week-1], [mm month-1] or similar.
- **op** (*{'wettest', 'driest'}*) – Operation to perform : 'wettest' calculate the wettest period ; 'driest' calculate the driest period.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [length] – Precipitation of {op} period

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices.prcptot_wetdry_quarter`(*pr: DataArray*, *op: Optional[str] = None*, *freq: str = 'YS'*) → DataArray

ANUCLIM Total precipitation of wettest/driest quarter.

The wettest (or driest) quarter of the year is determined, and the total precipitation of this period is calculated. If the input data frequency is daily ("D") or weekly ("W") quarters are defined as 13-week periods, otherwise are 3 months.

Parameters

- **pr** (*xarray.DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency.
- **op** (*{'wettest', 'driest'}*) – Operation to perform : 'wettest' calculate the wettest quarter ; 'driest' calculate the driest quarter.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [length] – Precipitation of {op} quarter

Examples

The following would compute for each grid cell of file *pr.day.nc* the annual wettest quarter total precipitation:

```
>>> from xclim.indices import prcptot_wetdry_quarter
>>> p = xr.open_dataset(path_to_pr_file)
>>> pr_warm_qrt = prcptot_wetdry_quarter(pr=p.pr, op="wettest")
```

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices.precip_accumulation`(*pr: xarray.DataArray*, *tas: xarray.DataArray = None*, *phase: str | None = None*, *thresh: str = '0 degC'*, *freq: str = 'YS'*) → xarray.DataArray

Accumulated total (liquid and/or solid) precipitation.

Resample the original daily mean precipitation flux and accumulate over each period. If a daily temperature is provided, the *phase* keyword can be used to sum precipitation of a given phase only. When the temperature is under the provided threshold, precipitation is assumed to be snow, and liquid rain otherwise. This indice is agnostic to the type of daily temperature (tas, tasmax or tasmin) given.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.

- **tas** (*xarray.DataArray, optional*) – Mean, maximum or minimum daily temperature.
- **phase** (*{None, 'liquid', 'solid'}*) – Which phase to consider, "liquid" or "solid", if None (default), both are considered.
- **thresh** (*str*) – Threshold of *tas* over which the precipication is assumed to be liquid rain.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [length] – The total daily precipitation at the given time frequency for the given phase.

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$

If tas and phase are given, the corresponding phase precipitation is estimated before computing the accumulation, using one of *snowfall_approximation* or *rain_approximation* with the *binary* method.

Examples

The following would compute, for each grid cell of a dataset, the total precipitation at the seasonal frequency, ie DJF, MAM, JJA, SON, DJF, etc.:

```
>>> from xclim.indices import precip_accumulation
>>> pr_day = xr.open_dataset(path_to_pr_file).pr
>>> prcp_tot_seasonal = precip_accumulation(pr_day, freq="QS-DEC")
```

`xclim.indices.precip_seasonality(`*pr: DataArray, freq: str = 'YS'*`)` → DataArray

ANUCLIM Precipitation Seasonality (C of V).

The annual precipitation Coefficient of Variation (C of V) expressed in percent. Calculated as the standard deviation of precipitation values for a given year expressed as a percentage of the mean of those values.

Parameters

- **pr** (*xarray.DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency. Units need to be defined as a rate (e.g. mm d-1, mm week-1).
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [%] – Precipitation coefficient of variation

Examples

The following would compute for each grid cell of file *pr.day.nc* the annual precipitation seasonality:

```
>>> import xclim.indices as xci
>>> p = xr.open_dataset(path_to_pr_file).pr
>>> pday_seasonality = xci.precip_seasonality(p)
>>> p_weekly = xci.precip_accumulation(p, freq="7D")
```

# Input units need to be a rate >>> p_weekly.attrs[“units”] = “mm/week” >>> pweek_seasonality = xci.precip_seasonality(p_weekly)

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

If input units are in mm s-1 (or equivalent) values are converted to mm/day to avoid potentially small denominator values.

`xclim.indices.qian_weighted_mean_average(`*tas: DataArray, dim: str = 'time'*`)` → DataArray

Binomial smoothed, five-day weighted mean average temperature.

Calculates a five-day weighted moving average with emphasis on temperatures closer to day of interest.

Parameters

- **tas** (*xr.DataArray*) – Daily mean temperature.
- **dim** (*str*) – Time dimension.

Returns

xr.DataArray – Binomial smoothed, five-day weighted mean average temperature.

Notes

Qian Modified Weighted Mean Indice originally proposed in [Qian&al2009]_, based on [BootsmaGameda&McKenney2005]_.

Let X_n be the average temperature for day n and X_t be the daily mean temperature on day t. Then the weighted mean average can be calculated as follows:

$$\overline{X}_n = \frac{X_{n-2} + 4X_{n-1} + 6X_n + 4X_{n+1} + X_{n+2}}{16}$$

References

`xclim.indices.rain_approximation`(*pr: DataArray*, *tas: DataArray*, *thresh: str = '0 degC'*, *method: str = 'binary'*) → DataArray

Rainfall approximation from total precipitation and temperature.

Liquid precipitation estimated from precipitation and temperature according to a given method. This is a convenience method based on `snowfall_approximation()`, see the latter for details.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **tas** (*xarray.DataArray, optional*) – Mean, maximum, or minimum daily temperature.
- **thresh** (*str,*) – Threshold temperature, used by method "binary".
- **method** (*{"binary", "brown", "auer"}*) – Which method to use when approximating snowfall from total precipitation. See notes.

Returns

xarray.DataArray, [same units as pr] – Liquid precipitation rate.

Notes

This method computes the snowfall approximation and subtracts it from the total precipitation to estimate the liquid rain precipitation.

See also:

`xclim.indices.snowfall_approximation()`

`xclim.indices.rain_on_frozen_ground_days`(*pr: DataArray*, *tas: DataArray*, *thresh: str = '1 mm/d'*, *freq: str = 'YS'*) → DataArray

Number of rain on frozen ground events.

Number of days with rain above a threshold after a series of seven days below freezing temperature. Precipitation is assumed to be rain when the temperature is above 0℃.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Precipitation threshold to consider a day as a rain event.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – The number of rain on frozen ground events per period.

Notes

Let PR_i be the mean daily precipitation and TG_i be the mean daily temperature of day i. Then for a period j, rain on frozen grounds days are counted where:

$$PR_i > Threshold[mm]$$

and where

$$TG_i 0$$

is true for continuous periods where $i 7$

`xclim.indices.rb_flashiness_index(`*q: DataArray*, *freq: str = 'YS'*`)` → DataArray

Richards-Baker flashiness index.

Measures oscillations in flow relative to total flow, quantifying the frequency and rapidity of short term changes in flow, based on Baker et al. (2004; [baker2004]).

Parameters

- **q** (*xarray.DataArray*) – Rate of river discharge.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – R-B Index.

Notes

Let $\mathbf{q} = q_0, q_1, \ldots, q_n$ be the sequence of daily discharge, the R-B Index is given by:

$$\frac{\sum_{i=1}^{n} |q_i - q_{i-1}|}{\sum_{i=1}^{n} q_i}$$

References

`xclim.indices.relative_humidity(`*tas: DataArray*, *tdps: Optional[DataArray] = None*, *huss: Optional[DataArray] = None*, *ps: Optional[DataArray] = None*, *ice_thresh: Optional[str] = None*, *method: str = 'sonntag90'*, *invalid_values: str = 'clip'*`)` → DataArray

Relative humidity.

Compute relative humidity from temperature and either dewpoint temperature or specific humidity and pressure through the saturation vapor pressure.

Parameters

- **tas** (*xr.DataArray*) – Temperature array
- **tdps** (*xr.DataArray*) – Dewpoint temperature, if specified, overrides huss and ps.
- **huss** (*xr.DataArray*) – Specific humidity.
- **ps** (*xr.DataArray*) – Air Pressure.
- **ice_thresh** (*str*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water. Does nothing if 'method' is "bohren98".

- **method** (*{"bohren98", "goffgratch46", "sonntag90", "tetens30", "wmo08"}*) – Which method to use, see notes of this function and of *saturation_vapor_pressure*.
- **invalid_values** (*{"clip", "mask", None}*) – What to do with values outside the 0-100 range. If "clip" (default), clips everything to 0 - 100, if "mask", replaces values outside the range by np.nan, and if *None*, does nothing.

Returns

xr.DataArray, [%] – Relative humidity.

Notes

In the following, let T, T_d, q and p be the temperature, the dew point temperature, the specific humidity and the air pressure.

For the "bohren98" method : This method does not use the saturation vapor pressure directly, but rather uses an approximation of the ratio of $\frac{e_{sat}(T_d)}{e_{sat}(T)}$. With L the enthalpy of vaporization of water and R_w the gas constant for water vapor, the relative humidity is computed as:

$$RH = e^{\frac{-L(T-T_d)}{R_w T T_d}}$$

From [BohrenAlbrecht1998], formula taken from [Lawrence2005]. $L = 2.5 \times 10^{-6}$ J kg-1, exact for $T = 273.15$ K, is used.

Other methods: With w, w_{sat}, e_{sat} the mixing ratio, the saturation mixing ratio and the saturation vapor pressure. If the dewpoint temperature is given, relative humidity is computed as:

$$RH = 100\frac{e_{sat}(T_d)}{e_{sat}(T)}$$

Otherwise, the specific humidity and the air pressure must be given so relative humidity can be computed as:

$$RH = 100\frac{w}{w_{sat}} w = \frac{q}{1-q} w_{sat} = 0.622\frac{e_{sat}}{P - e_{sat}}$$

The methods differ by how e_{sat} is computed. See the doc of `xclim.core.utils.saturation_vapor_pressure()`.

References

`xclim.indices.rprctot`(*pr: DataArray*, *prc: DataArray*, *thresh: str = '1.0 mm/day'*, *freq: str = 'YS'*) → DataArray

Proportion of accumulated precipitation arising from convective processes.

Return the proportion of total accumulated precipitation due to convection on days with total precipitation exceeding a specified threshold during the given period.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **prc** (*xarray.DataArray*) – Daily convective precipitation.
- **thresh** (*str*) – Precipitation value over which a day is considered wet.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – The proportion of the total precipitation accounted for by convective precipitation for each period.

`xclim.indices.saturation_vapor_pressure(`*tas: DataArray, ice_thresh: Optional[str] = None, method: str = 'sonntag90'*`)` → DataArray

Saturation vapor pressure from temperature.

Parameters

- **tas** (*xr.DataArray*) – Temperature array.
- **ice_thresh** (*str*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water.
- **method** (*{"goffgratch46", "sonntag90", "tetens30", "wmo08", "its90"}*) – Which method to use, see notes.

Returns

xarray.DataArray, [Pa] – Saturation vapor pressure.

Notes

In all cases implemented here $log(e_{sat})$ is an empirically fitted function (usually a polynomial) where coefficients can be different when ice is taken as reference instead of water. Available methods are:

- "goffgratch46" or "GG46", based on [goffgratch46], values and equation taken from [voemel].
- "sonntag90" or "SO90", taken from [sonntag90].
- "tetens30" or "TE30", based on [tetens30], values and equation taken from [voemel].
- "wmo08" or "WMO08", taken from [wmo08].
- "its90" or "ITS90", taken from [its90].

References

`xclim.indices.sea_ice_area(`*siconc: DataArray, areacello: DataArray, thresh: str = '15 pct'*`)` → DataArray

Total sea ice area.

Sea ice area measures the total sea ice covered area where sea ice concentration is above a threshold, usually set to 15%.

Parameters

- **siconc** (*xarray.DataArray*) – Sea ice concentration (area fraction).
- **areacello** (*xarray.DataArray*) – Grid cell area (usually over the ocean).
- **thresh** (*str*) – Minimum sea ice concentration for a grid cell to contribute to the sea ice extent.

Returns

xarray.DataArray, [length]^2 – Sea ice area.

Notes

To compute sea ice area over a subregion, first mask or subset the input sea ice concentration data.

References

What is the difference between sea ice area and extent

`xclim.indices.sea_ice_extent(`*siconc: DataArray*, *areacello: DataArray*, *thresh: str = '15 pct'*`)` → DataArray

Total sea ice extent.

Sea ice extent measures the *ice-covered* area, where a region is considered ice-covered if its sea ice concentration is above a threshold usually set to 15%.

Parameters

- **siconc** (*xarray.DataArray*) – Sea ice concentration (area fraction).
- **areacello** (*xarray.DataArray*) – Grid cell area.
- **thresh** (*str*) – Minimum sea ice concentration for a grid cell to contribute to the sea ice extent.

Returns

xarray.DataArray, [length]^2 – Sea ice extent.

Notes

To compute sea ice area over a subregion, first mask or subset the input sea ice concentration data.

References

What is the difference between sea ice area and extent

`xclim.indices.sfcwind_2_uas_vas(`*sfcWind: xr.DataArray*, *sfcWindfromdir: xr.DataArray*`)` → tuple[xr.DataArray, xr.DataArray]

Eastward and northward wind components from the wind speed and direction.

Compute the eastward and northward wind components from the wind speed and direction.

Parameters

- **sfcWind** (*xr.DataArray*) – Wind velocity
- **sfcWindfromdir** (*xr.DataArray*) – Direction from which the wind blows, following the meteorological convention where 360 stands for North.

Returns

- **uas** (*xr.DataArray, [m s-1]*) – Eastward wind velocity.
- **vas** (*xr.DataArray, [m s-1]*) – Northward wind velocity.

`xclim.indices.snd_max_doy(`*snd: DataArray*, *freq: str = 'AS-JUL'*`)` → DataArray

Maximum snow depth day of year.

Day of year when surface snow reaches its peak value. If snow depth is 0 over entire period, return NaN.

Parameters

- **snd** (*xarray.DataArray*) – Surface snow depth.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – The day of year at which snow depth reaches its maximum value.

`xclim.indices.snow_cover_duration(`*snd: DataArray*, *thresh: str = '2 cm'*, *freq: str = 'AS-JUL'*`)` → DataArray

Number of days with snow depth above a threshold.

Number of days where surface snow depth is greater or equal to given threshold. WARNING: The default *freq* is valid for the northern hemisphere.

Parameters

- **snd** (*xarray.DataArray*) – Surface snow thickness.
- **thresh** (*str*) – Threshold snow thickness.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Number of days where snow depth is greater or equal to threshold.

`xclim.indices.snow_depth(`*snd: DataArray*, *freq: str = 'YS'*`)` → DataArray

Mean of daily average snow depth.

Resample the original daily mean snow depth series by taking the mean over each period.

Parameters

- **snd** (*xarray.DataArray*) – Mean daily snow depth.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as snd] – The mean daily snow depth at the given time frequency

`xclim.indices.snow_melt_we_max(`*snw: DataArray*, *window: int = 3*, *freq: str = 'AS-JUL'*`)` → DataArray

Maximum snow melt.

The maximum snow melt over a given number of days expressed in snow water equivalent.

Parameters

- **snw** (*xarray.DataArray*) – Snow amount (mass per area).
- **window** (*int*) – Number of days during which the melt is accumulated.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – The maximum snow melt over a given number of days for each period. [mass/area].

`xclim.indices.snowfall_approximation(`*pr: DataArray*, *tas: DataArray*, *thresh: str = '0 degC'*, *method: str = 'binary'*`)` → DataArray

Snowfall approximation from total precipitation and temperature.

Solid precipitation estimated from precipitation and temperature according to a given method.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **tas** (*xarray.DataArray, optional*) – Mean, maximum, or minimum daily temperature.
- **thresh** (*str,*) – Threshold temperature, used by method "binary".
- **method** (*{"binary", "brown", "auer"}*) – Which method to use when approximating snowfall from total precipitation. See notes.

Returns

xarray.DataArray, [same units as pr] – Solid precipitation flux.

Notes

The following methods are available to approximate snowfall and are drawn from the Canadian Land Surface Scheme (CLASS, [Verseghy09]).

- `'binary'` : When the temperature is under the freezing threshold, precipitation is assumed to be solid. The method is agnostic to the type of temperature used (mean, maximum or minimum).
- `'brown'` : The phase between the freezing threshold goes from solid to liquid linearly over a range of 2°C over the freezing point.
- `'auer'` : The phase between the freezing threshold goes from solid to liquid as a degree six polynomial over a range of 6°C over the freezing point.

References

https://gitlab.com/cccma/classic/-/blob/master/src/atmosphericVarsCalc.f90

`xclim.indices.snw_max`(*snw: DataArray, freq: str = 'AS-JUL'*) → DataArray

Maximum snow amount.

The maximum daily snow amount.

Parameters

- **snw** (*xarray.DataArray*) – Snow amount (mass per area).
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – The maximum snow amount over a given number of days for each period. [mass/area].

`xclim.indices.snw_max_doy`(*snw: DataArray, freq: str = 'AS-JUL'*) → DataArray

Maximum snow amount day of year.

Day of year when surface snow amount reaches its peak value. If snow amount is 0 over entire period, return NaN.

Parameters

- **snw** (*xarray.DataArray*) – Surface snow amount.

- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray – The day of year at which snow amount reaches its maximum value.

`xclim.indices.specific_humidity`(*tas: DataArray*, *hurs: DataArray*, *ps: DataArray*, *ice_thresh: Optional[str] = None*, *method: str = 'sonntag90'*, *invalid_values: Optional[str] = None*) → DataArray

Specific humidity from temperature, relative humidity and pressure.

Specific humidity is the ratio between the mass of water vapour and the mass of moist air [WMO08].

Parameters

- **tas** (*xr.DataArray*) – Temperature array
- **hurs** (*xr.DataArray*) – Relative Humidity.
- **ps** (*xr.DataArray*) – Air Pressure.
- **ice_thresh** (*str*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water.
- **method** (*{"goffgratch46", "sonntag90", "tetens30", "wmo08"}*) – Which method to use, see notes of this function and of *saturation_vapor_pressure*.
- **invalid_values** (*{"clip", "mask", None}*) – What to do with values larger than the saturation specific humidity and lower than 0. If "clip" (default), clips everything to 0 - q_sat if "mask", replaces values outside the range by np.nan, if None, does nothing.

Returns
xarray.DataArray, [dimensionless] – Specific humidity.

Notes

In the following, let T, $hurs$ (in %) and p be the temperature, the relative humidity and the air pressure. With w, w_{sat}, e_{sat} the mixing ratio, the saturation mixing ratio and the saturation vapor pressure, specific humidity q is computed as:

$$w_{sat} = 0.622\frac{e_{sat}}{P - e_{sat}}w = w_{sat} * hurs/100q = w/(1 + w)$$

The methods differ by how e_{sat} is computed. See the doc of *xclim.core.utils.saturation_vapor_pressure*.

If *invalid_values* is not *None*, the saturation specific humidity q_{sat} is computed as:

$$q_{sat} = w_{sat}/(1 + w_{sat})$$

References

`xclim.indices.specific_humidity_from_dewpoint`(*tdps: DataArray*, *ps: DataArray*, *method: str = 'sonntag90'*) → DataArray

Specific humidity from dewpoint temperature and air pressure.

Specific humidity is the ratio between the mass of water vapour and the mass of moist air [WMO08].

Parameters

- **tdps** (*xr.DataArray*) – Dewpoint temperature array.
- **ps** (*xr.DataArray*) – Air pressure array.
- **method** (*{"goffgratch46", "sonntag90", "tetens30", "wmo08"}*) – Method to compute the saturation vapor pressure.

Returns

xarray.DataArray, [dimensionless] – Specific humidity.

Notes

If e is the water vapor pressure, and p the total air pressure, then specific humidity is given by

$$q = m_w e/(m_a(p - e) + m_w e)$$

where m_w and m_a are the molecular weights of water and dry air respectively. This formula is often written with $= m_w/m_a$, which simplifies to $q = e/(p - e(1-))$.

References

`xclim.indices.tas(`*tasmin: DataArray*, *tasmax: DataArray*`)` → DataArray

Average temperature from minimum and maximum temperatures.

We assume a symmetrical distribution for the temperature and retrieve the average value as Tg = (Tx + Tn) / 2

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum (daily) temperature
- **tasmax** (*xarray.DataArray*) – Maximum (daily) temperature

Returns

xarray.DataArray – Mean (daily) temperature [same units as tasmin]

`xclim.indices.temperature_seasonality(`*tas: DataArray*, *freq: str = 'YS'*`)` → DataArray

ANUCLIM temperature seasonality (coefficient of variation).

The annual temperature coefficient of variation expressed in percent. Calculated as the standard deviation of temperature values for a given year expressed as a percentage of the mean of those temperatures.

Parameters

- **tas** (*xarray.DataArray*) – Mean temperature at daily, weekly, or monthly frequency.
- **freq** (*str*) – Resampling frequency.

Returns

- *xarray.DataArray, [%]* – Mean temperature coefficient of variation
- **freq** (*str*) – Resampling frequency.

Examples

The following would compute for each grid cell of file *tas.day.nc* the annual temperature seasonality:

```
>>> import xclim.indices as xci
>>> t = xr.open_dataset(path_to_tas_file).tas
>>> tday_seasonality = xci.temperature_seasonality(t)
>>> t_weekly = xci.tg_mean(t, freq="7D")
>>> tweek_seasonality = xci.temperature_seasonality(t_weekly)
```

Notes

For this calculation, the mean in degrees Kelvin is used. This avoids the possibility of having to divide by zero, but it does mean that the values are usually quite small.

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices.tg10p`(*tas: DataArray*, *tas_per: DataArray*, *freq: str = 'YS'*, *bootstrap: bool = False*) → DataArray

Number of days with daily mean temperature below the 10th percentile.

Number of days with daily mean temperature below the 10th percentile.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **tas_per** (*xarray.DataArray*) – 10th percentile of daily mean temperature.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [time] – Count of days with daily mean temperature below the 10th percentile [days].

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tg10p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> tas_per = percentile_doy(tas, per=10).sel(percentiles=10)
>>> cold_days = tg10p(tas, tas_per)
```

`xclim.indices.tg90p`(*tas: DataArray*, *tas_per: DataArray*, *freq: str = 'YS'*, *bootstrap: bool = False*) → DataArray

Number of days with daily mean temperature over the 90th percentile.

Number of days with daily mean temperature over the 90th percentile.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **tas_per** (*xarray.DataArray*) – 90th percentile of daily mean temperature.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [time] – Count of days with daily mean temperature below the 10th percentile [days].

Notes

The 90th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tg90p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> tas_per = percentile_doy(tas, per=90).sel(percentiles=90)
>>> hot_days = tg90p(tas, tas_per)
```

`xclim.indices.tg_days_above`(*tas: DataArray*, *thresh: str = '10.0 degC'*, *freq: str = 'YS'*)

Number of days with tas above a threshold.

Number of days where daily mean temperature exceeds a threshold.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.

- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days where tas > threshold.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then counted is the number of days where:

$$TG_{ij} > Threshold[]$$

`xclim.indices.tg_days_below(`*tas: DataArray*, *thresh: str = '10.0 degC'*, *freq: str = 'YS'*`)`

Number of days with tas below a threshold.

Number of days where daily mean temperature is below a threshold.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days where tas < threshold.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then counted is the number of days where:

$$TG_{ij} < Threshold[]$$

`xclim.indices.tg_max(`*tas: DataArray*, *freq: str = 'YS'*`)` → DataArray

Highest mean temperature.

The maximum of daily mean temperature.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [same units as tas] – Maximum of daily minimum temperature.

Notes

Let TN_{ij} be the mean temperature at day i of period j. Then the maximum daily mean temperature for period j is:

$$TNx_j = max(TN_{ij})$$

`xclim.indices.tg_mean(`*tas: DataArray*, *freq: str = 'YS'*`)` → DataArray

Mean of daily average temperature.

Resample the original daily mean temperature series by taking the mean over each period.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tas] – The mean daily temperature at the given time frequency

Notes

Let TN_i be the mean daily temperature of day i, then for a period p starting at day a and finishing on day b:

$$TG_p = \frac{\sum_{i=a}^{b} TN_i}{b - a + 1}$$

Examples

The following would compute for each grid cell of file *tas.day.nc* the mean temperature at the seasonal frequency, ie DJF, MAM, JJA, SON, DJF, etc.:

```
>>> from xclim.indices import tg_mean
>>> t = xr.open_dataset(path_to_tas_file).tas
>>> tg = tg_mean(t, freq="QS-DEC")
```

`xclim.indices.tg_mean_warmcold_quarter(`*tas: DataArray*, *op: Optional[str] = None*, *freq: str = 'YS'*`)` → DataArray

ANUCLIM Mean temperature of warmest/coldest quarter.

The warmest (or coldest) quarter of the year is determined, and the mean temperature of this period is calculated. If the input data frequency is daily ("D") or weekly ("W"), quarters are defined as 13-week periods, otherwise as 3 months.

Parameters

- **tas** (*xarray.DataArray*) – Mean temperature at daily, weekly, or monthly frequency.
- **op** (*str {'warmest', 'coldest'}*) – Operation to perform: 'warmest' calculate the warmest quarter; 'coldest' calculate the coldest quarter.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same as tas] – Mean temperature of {op} quarter

Examples

The following would compute for each grid cell of file *tas.day.nc* the annual temperature warmest quarter mean temperature:

```
>>> import xclim.indices as xci
>>> t = xr.open_dataset(path_to_tas_file)
>>> t_warm_qrt = xci.tg_mean_warmcold_quarter(tas=t.tas, op="warmest")
```

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices.tg_mean_wetdry_quarter(`*tas: DataArray*, *pr: DataArray*, *op: Optional[str] = None*, *freq: str = 'YS'*`)` → DataArray

ANUCLIM Mean temperature of wettest/driest quarter.

The wettest (or driest) quarter of the year is determined, and the mean temperature of this period is calculated. If the input data frequency is daily ("D") or weekly ("W"), quarters are defined as 13-week periods, otherwise are 3 months.

Parameters

- **tas** (*xarray.DataArray*) – Mean temperature at daily, weekly, or monthly frequency.
- **pr** (*xarray.DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency.
- **op** (*{'wettest', 'driest'}*) – Operation to perform: 'wettest' calculate for the wettest quarter; 'driest' calculate for the driest quarter.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same as tas] – Mean temperature of {op} quarter

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices.tg_min(`*tas: DataArray*, *freq: str = 'YS'*`)` → DataArray

Lowest mean temperature.

Minimum of daily mean temperature.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tas] – Minimum of daily minimum temperature.

Notes

Let TG_{ij} be the mean temperature at day i of period j. Then the minimum daily mean temperature for period j is:

$$TGn_j = min(TG_{ij})$$

`xclim.indices.tn10p`(*tasmin: DataArray*, *tasmin_per: DataArray*, *freq: str = 'YS'*, *bootstrap: bool = False*) → DataArray

Number of days with daily minimum temperature below the 10th percentile.

Number of days with daily minimum temperature below the 10th percentile.

Parameters

- **tasmin** (*xarray.DataArray*) – Mean daily temperature.
- **tasmin_per** (*xarray.DataArray*) – 10th percentile of daily minimum temperature.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [time] – Count of days with daily minimum temperature below the 10th percentile [days].

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tn10p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> tas_per = percentile_doy(tas, per=10).sel(percentiles=10)
>>> cold_days = tn10p(tas, tas_per)
```

`xclim.indices.tn90p`(*tasmin: DataArray*, *tasmin_per: DataArray*, *freq: str = 'YS'*, *bootstrap: bool = False*) → DataArray

Number of days with daily minimum temperature over the 90th percentile.

Number of days with daily minimum temperature over the 90th percentile.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmin_per** (*xarray.DataArray*) – 90th percentile of daily minimum temperature.
- **freq** (*str*) – Resampling frequency.

- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns
xarray.DataArray, [time] – Count of days with daily minimum temperature below the 10th percentile [days].

Notes

The 90th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tn90p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> tas_per = percentile_doy(tas, per=90).sel(percentiles=90)
>>> hot_days = tn90p(tas, tas_per)
```

`xclim.indices.tn_days_above(`*tasmin: DataArray, thresh: str = '20.0 degC', freq: str = 'YS'*`)`

Number of days with tasmin above a threshold (number of tropical nights).

Number of days where daily minimum temperature exceeds a threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days where tasmin > threshold.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} > Threshold[]$$

`xclim.indices.tn_days_below(`*tasmin: DataArray, thresh: str = '-10.0 degC', freq: str = 'YS'*`)` → DataArray

Number of days with tasmin below a threshold.

Number of days where daily minimum temperature is below a threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.

- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days where tasmin < threshold.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} < Threshold[]$$

`xclim.indices.tn_max(`*tasmin: DataArray*, *freq: str = 'YS'*`)` → DataArray

Highest minimum temperature.

The maximum of daily minimum temperature.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [same units as tasmin] – Maximum of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then the maximum daily minimum temperature for period j is:

$$TNx_j = max(TN_{ij})$$

`xclim.indices.tn_mean(`*tasmin: DataArray*, *freq: str = 'YS'*`)` → DataArray

Mean minimum temperature.

Mean of daily minimum temperature.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [same units as tasmin] – Mean of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then mean values in period j are given by:

$$TN_{ij} = \frac{\sum_{i=1}^{I} TN_{ij}}{I}$$

`xclim.indices.tn_min(`*tasmin: DataArray*, *freq: str = 'YS'*`)` → DataArray

Lowest minimum temperature.

Minimum of daily minimum temperature.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tasmin] – Minimum of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then the minimum daily minimum temperature for period j is:

$$TNn_j = min(TN_{ij})$$

`xclim.indices.tropical_nights(`*tasmin: DataArray*, *thresh: str = '20.0 degC'*, *freq: str = 'YS'*`)` → DataArray

Tropical nights.

The number of days with minimum daily temperature above threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Number of days with minimum daily temperature above threshold.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} > Threshold[]$$

Warning: The *tropical_nights* indice is being deprecated in favour of *tn_days_above* with *thresh="20 degC"* by default. The indicator reflects this change. This indice will be removed in a future version of xclim.

`xclim.indices.tx10p(`*tasmax: DataArray*, *tasmax_per: DataArray*, *freq: str = 'YS'*, *bootstrap: bool = False*`)` → DataArray

Number of days with daily maximum temperature below the 10th percentile.

Number of days with daily maximum temperature below the 10th percentile.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **tasmax_per** (*xarray.DataArray*) – 10th percentile of daily maximum temperature.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [time] – Count of days with daily maximum temperature below the 10th percentile [days].

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tx10p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> tasmax_per = percentile_doy(tas, per=10).sel(percentiles=10)
>>> cold_days = tx10p(tas, tasmax_per)
```

`xclim.indices.tx90p(`*tasmax: DataArray, tasmax_per: DataArray, freq: str = 'YS', bootstrap: bool = False*`)` → DataArray

Number of days with daily maximum temperature over the 90th percentile.

Number of days with daily maximum temperature over the 90th percentile.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **tasmax_per** (*xarray.DataArray*) – 90th percentile of daily maximum temperature.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [time] – Count of days with daily maximum temperature below the 10th percentile [days].

Notes

The 90th percentile should be computed for a 5-day window centered on each calendar day for a reference period.

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tx90p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> tasmax_per = percentile_doy(tas, per=90).sel(percentiles=90)
>>> hot_days = tx90p(tas, tasmax_per)
```

`xclim.indices.tx_days_above(`*tasmax: DataArray, thresh: str = '25.0 degC', freq: str = 'YS'*`)` → DataArray

Number of days with tasmax above a threshold (number of summer days).

Number of days where daily maximum temperature exceeds a threshold.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Number of days where tasmax > threshold (number of summer days).

Notes

Let TX_{ij} be the daily maximum temperature at day i of period j. Then counted is the number of days where:

$$TX_{ij} > Threshold[]$$

`xclim.indices.tx_days_below(`*tasmax: DataArray, thresh: str = '25.0 degC', freq: str = 'YS'*`)`

Number of days with tmax below a threshold.

Number of days where daily maximum temperature is below a threshold.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Number of days where tasmin < threshold.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} < Threshold[]$$

`xclim.indices.tx_max(`*tasmax: DataArray*, *freq: str = 'YS'*`)` → DataArray

Highest max temperature.

The maximum value of daily maximum temperature.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tasmax] – Maximum value of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then the maximum daily maximum temperature for period j is:

$$TXx_j = max(TX_{ij})$$

`xclim.indices.tx_mean(`*tasmax: DataArray*, *freq: str = 'YS'*`)` → DataArray

Mean max temperature.

The mean of daily maximum temperature.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tasmax] – Mean of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then mean values in period j are given by:

$$TX_{ij} = \frac{\sum_{i=1}^{I} TX_{ij}}{I}$$

`xclim.indices.tx_min(`*tasmax: DataArray*, *freq: str = 'YS'*`)` → DataArray

Lowest max temperature.

The minimum of daily maximum temperature.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.

- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tasmax] – Minimum of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then the minimum daily maximum temperature for period j is:

$$TXn_j = min(TX_{ij})$$

`xclim.indices.tx_tn_days_above`(*tasmin: DataArray*, *tasmax: DataArray*, *thresh_tasmin: str = '22 degC'*, *thresh_tasmax: str = '30 degC'*, *freq: str = 'YS'*) → DataArray

Number of days with both hot maximum and minimum daily temperatures.

The number of days per period with tasmin above a threshold and tasmax above another threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – Threshold temperature for tasmin on which to base evaluation.
- **thresh_tasmax** (*str*) – Threshold temperature for tasmax on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – the number of days with tasmin > thresh_tasmin and tasmax > thresh_tasmax per period.

Notes

Let TX_{ij} be the maximum temperature at day i of period j, TN_{ij} the daily minimum temperature at day i of period j, TX_{thresh} the threshold for maximum daily temperature, and TN_{thresh} the threshold for minimum daily temperature. Then counted is the number of days where:

$$TX_{ij} > TX_{thresh}[]$$

and where:

$$TN_{ij} > TN_{thresh}[]$$

`xclim.indices.uas_vas_2_sfcwind`(*uas: xr.DataArray*, *vas: xr.DataArray*, *calm_wind_thresh: str = '0.5 m/s'*) → tuple[xr.DataArray, xr.DataArray]

Wind speed and direction from the eastward and northward wind components.

Computes the magnitude and angle of the wind vector from its northward and eastward components, following the meteorological convention that sets calm wind to a direction of 0° and northerly wind to 360°.

Parameters

- **uas** (*xr.DataArray*) – Eastward wind velocity
- **vas** (*xr.DataArray*) – Northward wind velocity
- **calm_wind_thresh** (*str*) – The threshold under which winds are considered "calm" and for which the direction is set to 0. On the Beaufort scale, calm winds are defined as < 0.5 m/s.

Returns

- **wind** (*xr.DataArray, [m s-1]*) – Wind velocity
- **wind_from_dir** (*xr.DataArray, [°]*) – Direction from which the wind blows, following the meteorological convention where 360 stands for North and 0 for calm winds.

Notes

Winds with a velocity less than *calm_wind_thresh* are given a wind direction of 0°, while stronger northerly winds are set to 360°.

`xclim.indices.universal_thermal_climate_index(`*tas: DataArray*, *hurs: DataArray*, *sfcWind: DataArray*, *mrt: Optional[DataArray] = None*, *rsds: Optional[DataArray] = None*, *rsus: Optional[DataArray] = None*, *rlds: Optional[DataArray] = None*, *rlus: Optional[DataArray] = None*, *stat: str = 'average'*, *mask_invalid: bool = True*`)` → DataArray

Universal thermal climate index.

The UTCI is the equivalent temperature for the environment derived from a reference environment and is used to evaluate heat stress in outdoor spaces.

Parameters

- **tas** (*xarray.DataArray*) – Mean temperature
- **hurs** (*xarray.DataArray*) – Relative Humidity
- **sfcWind** (*xarray.DataArray*) – Wind velocity
- **mrt** (*xarray.DataArray, optional*) – Mean radiant temperature
- **rsds** (*xr.DataArray, optional*) – Surface Downwelling Shortwave Radiation This is necessary if mrt is not None.
- **rsus** (*xr.DataArray, optional*) – Surface Upwelling Shortwave Radiation This is necessary if mrt is not None.
- **rlds** (*xr.DataArray, optional*) – Surface Downwelling Longwave Radiation This is necessary if mrt is not None.
- **rlus** (*xr.DataArray, optional*) – Surface Upwelling Longwave Radiation This is necessary if mrt is not None.
- **stat** (*{'average', 'instant', 'sunlit'}*) – Which statistic to apply. If "average", the average of the cosine of the solar zenith angle is calculated. If "instant", the instantaneous cosine of the solar zenith angle is calculated. If "sunlit", the cosine of the solar zenith angle is calculated during the sunlit period of each interval. If "instant", the instantaneous cosine of the solar zenith angle is calculated. This is necessary if mrt is not None.

- **mask_invalid** (*boolean*) – If True (default), UTCI values are NaN where any of the inputs are outside their validity ranges : -50°C < tas < 50°C, -30°C < tas - mrt < 30°C and 0.5 m/s < sfcWind < 17.0 m/s.

Returns

xarray.DataArray – Universal Thermal Climate Index.

Notes

The calculation uses water vapor partial pressure, which is derived from relative humidity and saturation vapor pressure computed according to the ITS-90 equation.

This code was inspired by the *pythermalcomfort* and *thermofeel* packages.

References

Bröde, Peter (2009). Program for calculating UTCI Temperature (UTCI), version a 0.002, http://www.utci.org/public/UTCI%20Program%20Code/UTCI_a002.f90 Błażejczyk, K., Jendritzky, G., Bröde, P., Fiala, D., Havenith, G., Epstein, Y., Psikuta, A., & Kampmann, B. (2013). An introduction to the Universal Thermal Climate Index (UTCI). DOI:10.7163/GPOL.2013.1

See also:

`http`
//www.utci.org/utcineu/utcineu.php

`xclim.indices.warm_and_dry_days(`*tas: DataArray*, *pr: DataArray*, *tas_per: DataArray*, *pr_per: DataArray*, *freq: str = 'YS'*`)` → DataArray

Warm and dry days.

Returns the total number of days where "warm" and "Dry" conditions coincide.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature values
- **pr** (*xarray.DataArray*) – Daily precipitation.
- **tas_per** (*xarray.DataArray*) – Third quartile of daily mean temperature computed by month.
- **pr_per** (*xarray.DataArray*) – First quartile of daily total precipitation computed by month.

Warning: Before computing the percentiles, all the precipitation below 1mm must be filtered out ! Otherwise, the percentiles will include non-wet days.

- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, – The total number of days where warm and dry conditions coincide.

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [warm_dry_days].

References

`xclim.indices.warm_and_wet_days(`*tas: DataArray*, *pr: DataArray*, *tas_per: DataArray*, *pr_per: DataArray*, *freq: str = 'YS'*`)` → DataArray

Warm and wet days.

Returns the total number of days where "warm" and "wet" conditions coincide.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature values
- **pr** (*xarray.DataArray*) – Daily precipitation.
- **tas_per** (*xarray.DataArray*) – Third quartile of daily mean temperature computed by month.
- **pr_per** (*xarray.DataArray*) – Third quartile of daily total precipitation computed by month.

  Warning: Before computing the percentiles, all the precipitation below 1mm must be filtered out ! Otherwise, the percentiles will include non-wet days.

- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, – The total number of days where warm and wet conditions coincide.

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [warm_wet_days].

References

`xclim.indices.warm_day_frequency(`*tasmax: DataArray*, *thresh: str = '30 degC'*, *freq: str = 'YS'*`)` → DataArray

Frequency of extreme warm days.

Return the number of days with tasmax > thresh per period

Parameters

- **tasmax** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days with tasmax > threshold per period.

Notes

Let TX_{ij} be the daily maximum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} > Threshold[]$$

`xclim.indices.warm_night_frequency(`*tasmin: DataArray*, *thresh: str = '22 degC'*, *freq: str = 'YS'*`)` → DataArray

Frequency of extreme warm nights.

Return the number of days with tasmin > thresh per period

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days with tasmin > threshold per period.

`xclim.indices.warm_spell_duration_index(`*tasmax: DataArray*, *tasmax_per: DataArray*, *window: int = 6*, *freq: str = 'YS'*, *bootstrap: bool = False*`)` → DataArray

Warm spell duration index.

Number of days inside spells of a minimum number of consecutive days where the daily maximum temperature is above the 90th percentile. The 90th percentile should be computed for a 5-day moving window, centered on each calendar day in the 1961-1990 period.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **tasmax_per** (*xarray.DataArray*) – percentile(s) of daily maximum temperature.
- **window** (*int*) – Minimum number of days with temperature above threshold to qualify as a warm spell.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns
xarray.DataArray, [time] – Warm spell duration index.

References

From the Expert Team on Climate Change Detection, Monitoring and Indices (ETCCDMI). Used in Alexander, L. V., et al. (2006), Global observed changes in daily climate extremes of temperature and precipitation, J. Geophys. Res., 111, D05109, doi: 10.1029/2005JD006290.

Examples

Note that this example does not use a proper 1961-1990 reference period.

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import warm_spell_duration_index
```

```
>>> tasmax = xr.open_dataset(path_to_tasmax_file).tasmax.isel(lat=0, lon=0)
>>> tasmax_per = percentile_doy(tasmax, per=90).sel(percentiles=90)
>>> warm_spell_duration_index(tasmax, tasmax_per)
```

`xclim.indices.water_budget`(*pr: xarray.DataArray*, *evspsblpot: xarray.DataArray | None = None*, *tasmin: xarray.DataArray | None = None*, *tasmax: xarray.DataArray | None = None*, *tas: xarray.DataArray | None = None*, *lat: xarray.DataArray | None = None*, *method: str = 'BR65'*) → xarray.DataArray

Precipitation minus potential evapotranspiration.

Precipitation minus potential evapotranspiration as a measure of an approximated surface water budget, where the potential evapotranspiration can be calculated with a given method.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **evspsblpot** (*xarray.DataArray*) – Potential evapotranspiration
- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **lat** (*xarray.DataArray*) – Latitude, needed if evspsblpot is not given.
- **method** (*str*) – Method to use to calculate the potential evapotranspiration.

Notes

Available methods are listed in the description of xclim.indicators.atmos.potential_evapotranspiration.

Returns

xarray.DataArray, – Precipitation minus potential evapotranspiration.

`xclim.indices.wetdays`(*pr: DataArray*, *thresh: str = '1.0 mm/day'*, *freq: str = 'YS'*) → DataArray

Wet days.

Return the total number of days during period with precipitation over threshold.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.

- **thresh** (*str*) – Precipitation value over which a day is considered wet.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – The number of wet days for each period [day].

Examples

The following would compute for each grid cell of file *pr.day.nc* the number days with precipitation over 5 mm at the seasonal frequency, ie DJF, MAM, JJA, SON, DJF, etc.:

```
>>> from xclim.indices import wetdays
>>> pr = xr.open_dataset(path_to_pr_file).pr
>>> wd = wetdays(pr, thresh="5 mm/day", freq="QS-DEC")
```

`xclim.indices.wetdays_prop(`*pr: DataArray*, *thresh: str = '1.0 mm/day'*, *freq: str = 'YS'*`)` → DataArray

Proportion of wet days.

Return the proportion of days during period with precipitation over threshold.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **thresh** (*str*) – Precipitation value over which a day is considered wet.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – The proportion of wet days for each period [1].

Examples

The following would compute for each grid cell of file *pr.day.nc* the proportion of days with precipitation over 5 mm at the seasonal frequency, ie DJF, MAM, JJA, SON, DJF, etc.:

```
>>> from xclim.indices import wetdays_prop
>>> pr = xr.open_dataset(path_to_pr_file).pr
>>> wd = wetdays_prop(pr, thresh="5 mm/day", freq="QS-DEC")
```

`xclim.indices.wind_chill_index(`*tas: DataArray*, *sfcWind: DataArray*, *method: str = 'CAN'*, *mask_invalid: bool = True*`)`

Wind chill index.

The Wind Chill Index is an estimation of how cold the weather feels to the average person. It is computed from the air temperature and the 10-m wind. As defined by the Environment and Climate Change Canada ([MVSZ2015]), two equations exist, the conventional one and one for slow winds (usually $<$ 5 km/h), see Notes.

Parameters

- **tas** (*xarray.DataArray*) – Surface air temperature.
- **sfcWind** (*xarray.DataArray*) – Surface wind speed (10 m).
- **method** (*{'CAN', 'US'}*) – If "CAN" (default), a "slow wind" equation is used where winds are slower than 5 km/h, see Notes.

- **mask_invalid** (*bool*) – Whether to mask values when the inputs are outside their validity range. or not. If True (default), points where the temperature is above a threshold are masked. The threshold is 0°C for the canadian method and 50°F for the american one. With the latter method, points where sfcWind < 3 mph are also masked.

Returns
xarray.DataArray, [degC] – Wind Chill Index.

Notes

Following the calculations of Environment and Climate Change Canada, this function switches from the standardized index to another one for slow winds. The standard index is the same as used by the National Weather Service of the USA ([NWS]). Given a temperature at surface T (in °C) and 10-m wind speed V (in km/h), the Wind Chill Index W (dimensionless) is computed as:

$$W = 13.12 + 0.6125 * T - 11.37 * V^0.16 + 0.3965 * T * V^0.16$$

Under slow winds ($V < 5$ km/h), and using the canadian method, it becomes:

$$W = T + \frac{-1.59 + 0.1345 * T}{5} * V$$

Both equations are invalid for temperature over 0°C in the canadian method.

The american Wind Chill Temperature index (WCT), as defined by USA's National Weather Service, is computed when *method='US'*. In that case, the maximal valid temperature is 50°F (10 °C) and minimal wind speed is 3 mph (4.8 km/h).

See also:

`National`

References

`xclim.indices.windy_days(`*sfcWind: DataArray, thresh: str = '10.8 m s-1', freq: str = 'MS'*`)` → DataArray

Windy days.

The number of days with average near-surface wind speed above threshold.

Parameters

- **sfcWind** (*xarray.DataArray*) – Daily average near-surface wind speed.
- **thresh** (*str*) – Threshold average near-surface wind speed on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days with average near-surface wind speed above threshold.

Notes

Let WS_{ij} be the windspeed at day i of period j. Then counted is the number of days where:

$$WS_{ij} >= Threshold[ms - 1]$$

`xclim.indices.winter_rain_ratio(`**, pr: DataArray, prsn: Optional[DataArray] = None, tas: Optional[DataArray] = None, freq: str = 'QS-DEC'*`)` → DataArray

Ratio of rainfall to total precipitation during winter.

The ratio of total liquid precipitation over the total precipitation over the winter months (DJF. If solid precipitation is not provided, then precipitation is assumed solid if the temperature is below 0°C.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **prsn** (*xarray.DataArray, optional*) – Mean daily solid precipitation flux.
- **tas** (*xarray.DataArray, optional*) – Mean daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – Ratio of rainfall to total precipitation during winter months (DJF).

`xclim.indices.winter_storm(`*snd: DataArray, thresh: str = '25 cm', freq: str = 'AS-JUL'*`)` → DataArray

Days with snowfall over threshold.

Number of days with snowfall accumulation greater or equal to threshold.

Parameters

- **snd** (*xarray.DataArray*) – Surface snow depth.
- **thresh** (*str*) – Threshold on snowfall accumulation require to label an event a *winter storm*.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – Number of days per period identified as winter storms.

Notes

Snowfall accumulation is estimated by the change in snow depth.

14.2.2 Indices submodules

Fire Weather Indices Submodule

This submodule defines the `xclim.indices.fire_season()`, `xclim.indices.drought_code()` and `xclim.indices.fire_weather_indexes()` indices, which are used by the eponym indicators. Users should read this module's documentation and the one of *fire_ weather_ ufunc*.

First adapted from Matlab code *CalcFWITimeSeriesWithStartup.m* from GFWED made for using MERRA2 data, which was a translation of FWI.vba of the Canadian Fire Weather Index system. Then, updated and synchronized with the R code of the cffdrs package. When given the correct parameters, the current code has an error below 3% when compared with the [GFWED2015] data.

Parts of the code and of the documentation in this submodule are directly taken from [cffdrs] which was published with the GPLv2 license.

Fire season

Fire weather indexes are iteratively computed, each day's value depending on the previous day indexes. Additionally and optionally, the codes are "shut down" (set to NaN) in winter. There are a few ways of computing this shut down and the subsequent spring start-up. The *fire_ season* function allows for full control of that, replicating the *fireSeason* method in the R package. It produces a mask to be given a *season_ mask* in the indicators. However, the *fire_ weather_ ufunc* and the indicators also accept a *season_ method* parameter so the fire season can be computed inside the iterator. Passing *season_ method=None* switches to an "always on" mode replicating the *fwi* method of the R package.

The fire season determination is based on three consecutive daily maximum temperature thresholds ([Wotton&Flannigan1993]_, [Lawson&Armitage2008]_). A "GFWED" method is also implemented. There, the 12h LST temperature is used instead of the daily maximum. The current implementation is slightly different from the description in [GFWED2015], but it replicates the Matlab code when *temp_ start_ thresh* and *temp_ end_ thresh* are both set to 6 degC. In xclim, the number of consecutive days, the start and end temperature thresholds and the snow depth threshold can all be modified.

Overwintering

Additionnaly, overwintering of the drought code is also directly implemented in *fire_weather_ufunc()*. The last drought_code of the season is kept in "winter" (where the fire season mask is False) and the precipitation is accumulated until the start of the next season. The first drought code is computed as a function of these instead of using the default DCStart value. Parameters to `_overwintering_drought_code()` are listed below. The code for the overwintering is based on [MBHFJ2020].

Finally, a mechanism for dry spring starts is implemented. For now, it is slightly different from what the GFWED, uses, but seems to agree with the state of the science of the CFS. When activated, the drought code and Duff-moisture codes are started in spring with a value that is function of the number of days since the last significative precipitation event. The conventional start value increased by that number of days times a "dry start" factor. Parameters are controlled in the call of the indices and *fire_weather_ufunc()*. Overwintering of the drought code overrides this mechanism if both are activated. GFWED use a more complex approach with an added check on the previous day's snow cover for determining "dry" points. Moreover, there, the start values are only the multiplication of a factor to the number of dry days.

Examples

The current litterature seems to agree that climate-oriented series of the fire weather indexes should be computed using only the longest fire season of each year and activatting the overwintering of the drought code and the "dry start" for the duff-moisture code. The following example uses reasonable parameters when computing over all of Canada.

Note: Here the example snippets use the _indices_ defined in this very module, but we always recommend using the _indicators_ defined in the *xc.atmos* module.

```
>>> ds = open_dataset("ERA5/daily_surface_cancities_1990-1993.nc")
>>> ds = ds.assign(
...     hurs=xclim.atmos.relative_humidity_from_dewpoint(ds=ds),
...     tas=xclim.core.units.convert_units_to(ds.tas, "degC"),
...     pr=xclim.core.units.convert_units_to(ds.pr, "mm/d"),
...     sfcWind=xclim.atmos.wind_speed_from_vector(ds=ds)[0],
... )
>>> season_mask = fire_season(
...     tas=ds.tas,
...     method="WF93",
...     freq="YS",
...     # Parameters below are at their default values, but listed here for explicitness.
...     temp_start_thresh="12 degC",
...     temp_end_thresh="5 degC",
...     temp_condition_days=3,
... )
>>> out_fwi = fire_weather_indexes(
...     tas=ds.tas,
...     pr=ds.pr,
...     hurs=ds.hurs,
...     sfcWind=ds.sfcWind,
...     lat=ds.lat,
...     season_mask=season_mask,
...     overwintering=True,
...     dry_start="CFS",
...     prec_thresh="1.5 mm/d",
...     dmc_dry_factor=1.2,
...     # Parameters below are at their default values, but listed here for explicitness.
...     carry_over_fraction=0.75,
...     wetting_efficiency_fraction=0.75,
...     dc_start=15,
...     dmc_start=6,
...     ffmc_start=85,
... )
```

Similarly, the next lines calculate the fire weather indexes, but according to the parameters and options used in NASA's GFWED datasets. Here, no need to split the fire season mask from the rest of the computation as _all_ seasons are used, even the very short shoulder seasons.

```
>>> ds = open_dataset("FWI/GFWED_sample_2017.nc")
>>> out_fwi = fire_weather_indexes(
...     tas=ds.tas,
...     pr=ds.prbc,
...     snd=ds.snow_depth,
...     hurs=ds.rh,
...     sfcWind=ds.sfcwind,
...     lat=ds.lat,
...     season_method="GFWED",
...     overwintering=False,
...     dry_start="GFWED",
...     temp_start_thresh="6 degC",
...     temp_end_thresh="6 degC",
...     # Parameters below are at their default values, but listed here for explicitness.
```

(continues on next page)

(continued from previous page)

```
...     temp_condition_days=3,
...     snow_condition_days=3,
...     dc_start=15,
...     dmc_start=6,
...     ffmc_start=85,
...     dmc_dry_factor=2,
... )
```

References

Codes:

Updated source code for calculating fire danger indexes in the Canadian Forest Fire Weather Index System, Y. Wang, K.R. Anderson, and R.M. Suddaby, INFORMATION REPORT NOR-X-424, 2015.

https://cwfis.cfs.nrcan.gc.ca/background/dsm/fwi

Matlab code of the GFWED obtained through personal communication.

Fire season determination methods:

Drought Code overwintering:

`xclim.indices.fwi.drought_code(`*tas: xr.DataArray*, *pr: xr.DataArray*, *lat: xr.DataArray*, *snd: xr.DataArray | None = None*, *dc0: xr.DataArray | None = None*, *season_mask: xr.DataArray | None = None*, *season_method: str | None = None*, *overwintering: bool = False*, *dry_start: str | None = None*, *initial_start_up: bool = True*, ***params*`)`

Drought code (FWI component).

The drought code is part of the Canadian Forest Fire Weather Index System. It is a numeric rating of the average moisture content of organic layers.

Parameters

- **tas** (*xr.DataArray*) – Noon temperature.
- **pr** (*xr.DataArray*) – Rain fall in open over previous 24 hours, at noon.
- **lat** (*xr.DataArray*) – Latitude coordinate
- **snd** (*xr.DataArray*) – Noon snow depth.
- **dc0** (*xr.DataArray*) – Initial values of the drought code.
- **season_mask** (*xr.DataArray, optional*) – Boolean mask, True where/when the fire season is active.
- **season_method** (*{None, "WF93", "LA08", "GFWED"}*) – How to compute the start-up and shutdown of the fire season. If "None", no start-ups or shutdowns are computed, similar to the R fwi function. Ignored if *season_mask* is given.
- **overwintering** (*bool*) – Whether to activate DC overwintering or not. If True, either season_method or season_mask must be given.
- **dry_start** (*{None, "CFS", 'GFWED'}*) – Whether to activate the DC and DMC "dry start" mechanism and which method to use. , see `fire_weather_ufunc()`.

- **initial_start_up** (*bool*) – If True (default), grid points where the fire season is active on the first timestep go through a start_up phase for that time step. Otherwise, previous codes must be given as a continuing fire season is assumed for those points.
- **params** – Any other keyword parameters as defined in *xclim.indices.fwi.fire_weather_ufunc* and in `default_params`.

Returns
xr.DataArray, [dimensionless] – Drought code

Notes

See https://cwfis.cfs.nrcan.gc.ca/background/dsm/fwi, the module's doc and doc of `fire_weather_ufunc()` for more information.

References

Updated source code for calculating fire danger indexes in the Canadian Forest Fire Weather Index System, Y. Wang, K.R. Anderson, and R.M. Suddaby, INFORMATION REPORT NOR-X-424, 2015.

`xclim.indices.fwi.fire_season`(*tas: xr.DataArray*, *snd: xr.DataArray | None = None*, *method: str = 'WF93'*, *freq: str | None = None*, *temp_start_thresh: str = '12 degC'*, *temp_end_thresh: str = '5 degC'*, *temp_condition_days: int = 3*, *snow_condition_days: int = 3*, *snow_thresh: str = '0.01 m'*)

Fire season mask.

Binary mask of the active fire season, defined by conditions on consecutive daily temperatures and, optionally, snow depths.

Parameters

- **tas** (*xr.DataArray*) – Daily surface temperature, cffdrs recommends using maximum daily temperature.
- **snd** (*xr.DataArray, optional*) – Snow depth, used with method == 'LA08'.
- **method** (*{"WF93", "LA08", "GFWED"}*) – Which method to use. "LA08" and "GFWED" need the snow depth.
- **freq** (*str, optional*) – If given only the longest fire season for each period defined by this frequency, Every "seasons" are returned if None, including the short shoulder seasons.
- **temp_start_thresh** (*str*) – Minimal temperature needed to start the season.
- **temp_end_thresh** (*str*) – Maximal temperature needed to end the season.
- **temp_condition_days** (*int*) – Number of days with temperature above or below the thresholds to trigger a start or an end of the fire season.
- **snow_condition_days** (*int*) – Parameters for the fire season determination. See `fire_season()`. Temperature is in degC, snow in m. The *snow_thresh* parameters is also used when *dry_start* is set to "GFWED".
- **snow_thresh** (*str*) – Minimal snow depth level to end a fire season, only used with method "LA08".

Returns
fire_season (*xr.DataArray*) – Fire season mask

References

[Wotton&Flannigan1993]_

[Lawson&Armitage2008]_

`xclim.indices.fwi.fire_weather_indexes(`*tas: xr.DataArray, pr: xr.DataArray, sfcWind: xr.DataArray, hurs: xr.DataArray, lat: xr.DataArray, snd: xr.DataArray | None = None, ffmc0: xr.DataArray | None = None, dmc0: xr.DataArray | None = None, dc0: xr.DataArray | None = None, season_mask: xr.DataArray | None = None, season_method: str | None = None, overwintering: bool = False, dry_start: str | None = None, initial_start_up: bool = True, **params*`)`

Fire weather indexes.

Computes the 6 fire weather indexes as defined by the Canadian Forest Service: the Drought Code, the Duff-Moisture Code, the Fine Fuel Moisture Code, the Initial Spread Index, the Build Up Index and the Fire Weather Index.

Parameters

- **tas** (*xr.DataArray*) – Noon temperature.
- **pr** (*xr.DataArray*) – Rain fall in open over previous 24 hours, at noon.
- **sfcWind** (*xr.DataArray*) – Noon wind speed.
- **hurs** (*xr.DataArray*) – Noon relative humidity.
- **lat** (*xr.DataArray*) – Latitude coordinate
- **snd** (*xr.DataArray*) – Noon snow depth, only used if *season_method='LA08'* is passed.
- **ffmc0** (*xr.DataArray*) – Initial values of the fine fuel moisture code.
- **dmc0** (*xr.DataArray*) – Initial values of the Duff moisture code.
- **dc0** (*xr.DataArray*) – Initial values of the drought code.
- **season_mask** (*xr.DataArray, optional*) – Boolean mask, True where/when the fire season is active.
- **season_method** (*{None, "WF93", "LA08", "GFWED"}*) – How to compute the start-up and shutdown of the fire season. If "None", no start-ups or shutdowns are computed, similar to the R fwi function. Ignored if *season_mask* is given.
- **overwintering** (*bool*) – Whether to activate DC overwintering or not. If True, either season_method or season_mask must be given.
- **dry_start** (*{None, 'CFS', 'GFWED'}*) – Whether to activate the DC and DMC "dry start" mechanism or not, see `fire_weather_ufunc()`.
- **initial_start_up** (*bool*) – If True (default), gridpoints where the fire season is active on the first timestep go through a start_up phase for that time step. Otherwise, previous codes must be given as a continuing fire season is assumed for those points.
- **params** – Any other keyword parameters as defined in `fire_weather_ufunc()` and in `default_params`.

Returns

- **DC** (*xr.DataArray, [dimensionless]*)

- **DMC** (*xr.DataArray, [dimensionless]*)
- **FFMC** (*xr.DataArray, [dimensionless]*)
- **ISI** (*xr.DataArray, [dimensionless]*)
- **BUI** (*xr.DataArray, [dimensionless]*)
- **FWI** (*xr.DataArray, [dimensionless]*)

Notes

See https://cwfis.cfs.nrcan.gc.ca/background/dsm/fwi, the module's doc and doc of `fire_weather_ufunc()` for more information.

References

Updated source code for calculating fire danger indexes in the Canadian Forest Fire Weather Index System, Y. Wang, K.R. Anderson, and R.M. Suddaby, INFORMATION REPORT NOR-X-424, 2015.

`xclim.indices.fwi.fire_weather_ufunc(`**, tas: xr.DataArray, pr: xr.DataArray, hurs: xr.DataArray | None = None, sfcWind: xr.DataArray | None = None, snd: xr.DataArray | None = None, lat: xr.DataArray | None = None, dc0: xr.DataArray | None = None, dmc0: xr.DataArray | None = None, ffmc0: xr.DataArray | None = None, winter_pr: xr.DataArray | None = None, season_mask: xr.DataArray | None = None, start_dates: str | xr.DataArray | None = None, indexes: Sequence[str] = None, season_method: str | None = None, overwintering: bool = False, dry_start: str | None = None, initial_start_up: bool = True, **params*`)`

Fire Weather Indexes computation using xarray's apply_ufunc.

No unit handling. Meant to be used by power users only. Please prefer using the `DC` and `FWI` indicators or the `drought_code()` and `fire_weather_indexes()` indices defined in the same submodule.

Dask arrays must have only one chunk along the "time" dimension. User can control which indexes are computed with the *indexes* argument.

Parameters

- **tas** (*xr.DataArray*) – Noon surface temperature in °C
- **pr** (*xr.DataArray*) – Rainfall over previous 24h, at noon in mm/day
- **hurs** (*xr.DataArray, optional*) – Noon surface relative humidity in %, not needed for DC
- **sfcWind** (*xr.DataArray, optional*) – Noon surface wind speed in km/h, not needed for DC, DMC or BUI
- **snd** (*xr.DataArray, optional*) – Noon snow depth in m, only needed if *season_method* is "LA08"
- **lat** (*xr.DataArray, optional*) – Latitude in °N, not needed for FFMC or ISI
- **dc0** (*xr.DataArray, optional*) – Previous DC map, see Notes. Defaults to NaN.
- **dmc0** (*xr.DataArray, optional*) – Previous DMC map, see Notes. Defaults to NaN.
- **ffmc0** (*xr.DataArray, optional*) – Previous FFMC map, see Notes. Defaults to NaN.

- **winter_pr** (*xr.DataArray, optional*) – Accumulated precipitation since the end of the last season, until the beginning of the current data, mm/day. Only used if *overwintering* is True, defaults to 0.
- **season_mask** (*xr.DataArray, optional*) – Boolean mask, True where/when the fire season is active.
- **indexes** (*Sequence[str], optional*) – Which indexes to compute. If intermediate indexes are needed, they will be added to the list and output.
- **season_method** (*{None, "WF93", "LA08", "GFWED"}*) – How to compute the start-up and shutdown of the fire season. If "None", no start-ups or shutdowns are computed, similar to the R fwi function. Ignored if *season_mask* is given.
- **overwintering** (*bool*) – Whether to activate DC overwintering or not. If True, either season_method or season_mask must be given.
- **dry_start** (*{None, 'CFS', 'GFWED'}*) – Whether to activate the DC and DMC "dry start" mechanism and which method to use. See Notes. If overwintering is activated, it overrides this parameter : only DMC is handled through the dry start mechanism.
- **initial_start_up** (*bool*) – If True (default), grid points where the fire season is active on the first timestep go through a start-up phase for that time step. Otherwise, previous codes must be given as a continuing fire season is assumed for those points.
- **carry_over_fraction** (*float*)
- **wetting_efficiency_fraction** (*float*) – Drought code overwintering parameters, see `overwintering_drought_code()`.
- **temp_start_thresh** (*float*) – Starting temperature threshold.
- **temp_end_thresh** (*float*) – Ending temperature threshold.
- **temp_condition_days** (*int*) – The number of days' temperature condition to consider.
- **snow_thresh** (*float*)
- **snow_condition_days** (*int*) – Parameters for the fire season determination. See `fire_season()`. Temperature is in degC, snow in m. The *snow_thresh* parameters is also used when *dry_start* is set to "GFWED", see Notes.
- **dc_start** (*float*)
- **dmc_start** (*float*)
- **ffmc_start** (*float*) – Default starting values for the three base codes.
- **prec_thresh** (*float*) – If the "dry start" is activated, this is the "wet" day precipitation threshold, see Notes. In mm/d.
- **dc_dry_factor** (*float*) – DC's start-up values for the "dry start" mechanism, see Notes.
- **dmc_dry_factor** (*float*) – DMC's start-up values for the "dry start" mechanism, see Notes.
- **snow_cover_days** (*int*)
- **snow_min_cover_frac** (*float*)
- **snow_min_mean_depth** (*float*) – Additional parameters for GFWED's version of the "dry start" mechanism. See Notes. Snow depth is in m.

Returns

dict[str, xarray.DataArray] – Dictionary containing the computed indexes as prescribed in *indexes*, including the intermediate ones needed, even if they were not explicitly listed in *indexes*. When overwintering is activated, *winter_pr* is added. If *season_method* is not None and *season_mask* was not given, *season_mask* is computed on-the-fly and added to the output.

Notes

When overwintering is activated, the argument *dc0* is understood as last season's last DC map and will be used to compute the overwintered DC at the beginning of the next season.

If overwintering is not activated and neither is fire season computation (*season_method* and *season_mask* are *None*), *dc0*, *dmc0* and *ffmc0* are understood as the codes on the day before the first day of FWI computation. They will default to their respective start values. This "always on" mode replicates the R "fwi" code.

If the "dry start" mechanism is set to "CFS" (but there is no overwintering), the arguments *dc0* and *dmc0* are understood as the potential start-up values from last season. With DC_{start} the conventional start-up value, F_{dry-dc} the *dc_dry_factor* and N_{dry} the number of days since the last significative precipitation event, the start-up value DC_0 is computed as:

$$DC_0 = DC_{start} + F_{dry-dc} * N_{dry}$$

The last significative precipitation event is the last day where precipitation was greater or equal to "prec_thresh". The same happens for the DMC, with corresponding parameters. If overwintering is activated, this mechanism is only used for the DMC.

Alternatively, *dry_start* can be set to "GFWED". In this mode, the start-up values are computed as:

$$DC_0 = F_{dry-dc} * N_{dry}$$

Where the current day is also included in the determination of N_{dry} (DC_0 can thus be 0). Finally, for this "GFWED" mode, if snow cover is provided, a second check is performed: the dry start procedure is skipped and conventional start-up values are used for cells where the snow cover of the last *snow_cover_days* was above *snow_thresh* for at least *snow_cover_days* * *snow_min_cover_frac* days and where the mean snow cover over the same period was greater of equal to *snow_min_mean_depth*.

`xclim.indices.fwi.overwintering_drought_code`(*last_dc: xr.DataArray*, *winter_pr: xr.DataArray*, *carry_over_fraction: xr.DataArray | float = 0.75*, *wetting_efficiency_fraction: xr.DataArray | float = 0.75*, *min_dc: xr.DataArray | float = 15*) → xr.DataArray

Compute the season-starting drought code based on the previous season's last drought code and the total winter precipitation.

This method replicates the "wDC" method of the [cffdrs] R package, with an added control on the "minimum" DC.

Parameters

- **last_dc** (*xr.DataArray*) – The previous season's last drought code.
- **winter_pr** (*xr.DataArray*) – The accumulated precipitation since the end of the fire season.
- **carry_over_fraction** (*xr.DataArray or float*) – Carry-over fraction of last fall's moisture

- **wetting_efficiency_fraction** (*xr.DataArray or float*) – Effectiveness of winter precipitation in recharging moisture reserves in spring
- **min_dc** (*xr.DataArray or float*) – Minimum drought code starting value.

Returns
wDC (*xr.DataArray*) – Overwintered drought code.

Notes

Details taken from the R package documentation ([cffdrs]): Of the three fuel moisture codes (i.e. FFMC, DMC and DC) making up the FWI System, only the DC needs to be considered in terms of its values carrying over from one fire season to the next. In Canada both the FFMC and the DMC are assumed to reach moisture saturation from overwinter precipitation at or before spring melt; this is a reasonable assumption and any error in these assumed starting conditions quickly disappears. If snowfall (or other overwinter precipitation) is not large enough however, the fuel layer tracked by the Drought Code may not fully reach saturation after spring snow melt; because of the long response time in this fuel layer (53 days in standard conditions) a large error in this spring starting condition can affect the DC for a significant portion of the fire season. In areas where overwinter precipitation is 200 mm or more, full moisture recharge occurs and DC overwintering is usually unnecessary. More discussion of overwintering and fuel drying time lag can be found in [Lawson&Armitage2008]_ and [VanWagner1985].

Carry-over fraction of last fall's moisture:

- 1.0, Daily DC calculated up to 1 November; continuous snow cover, or freeze-up, whichever comes first
- 0.75, Daily DC calculations stopped before any of the above conditions met or the area is subject to occasional winter chinook conditions, leaving the ground bare and subject to moisture depletion
- 0.5, Forested areas subject to long periods in fall or winter that favor depletion of soil moisture

Effectiveness of winter precipitation in recharging moisture reserves in spring:

- 0.9, Poorly drained, boggy sites with deep organic layers
- 0.75, Deep ground frost does not occur until late fall, if at all; moderately drained sites that allow infiltration of most of the melting snowpack
- 0.5, Chinook-prone areas and areas subject to early and deep ground frost; well-drained soils favoring rapid percolation or topography favoring rapid runoff before melting of ground frost

Source: [Lawson&Armitage2008]_ - Table 9.

References

[cffdrs]

[Lawson&Armitage2008]_

[VanWagner1985]

Generic indices submodule

Helper functions for common generic actions done in the computation of indices.

`xclim.indices.generic.aggregate_between_dates(`*data: xr.DataArray*, *start: xr.DataArray | DayOfYearStr*, *end: xr.DataArray | DayOfYearStr*, *op: str = 'sum'*, *freq: str | None = None*`)` → xr.DataArray

Aggregate the data over a period between start and end dates and apply the operator on the aggregated data.

Parameters

- **data** (*xr.DataArray*) – Data to aggregate between start and end dates.
- **start** (*xr.DataArray or DayOfYearStr*) – Start dates (as day-of-year) for the aggregation periods.
- **end** (*xr.DataArray or DayOfYearStr*) – End (as day-of-year) dates for the aggregation periods.
- **op** (*{'min', 'max', 'sum', 'mean', 'std'}*) – Operator.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Aggregated data between the start and end dates. If the end date is before the start date, returns np.nan. If there is no start and/or end date, returns np.nan.

`xclim.indices.generic.compare(`*da: xr.DataArray*, *op: str*, *thresh: float | int*`)` → xr.DataArray

Compare a dataArray to a threshold using given operator.

Parameters

- **da** (*xr.DataArray*) – Input data.
- **op** (*{">", "<", ">=", "<=", "gt", "lt", "ge", "le"}*) – Logical operator {>, <, >=, <=, gt, lt, ge, le }. e.g. arr > thresh.
- **thresh** (*Union[float, int]*) – Threshold value.

Returns

xr.DataArray – Boolean mask of the comparison.

`xclim.indices.generic.count_level_crossings(`*low_data: DataArray*, *high_data: DataArray*, *threshold: str*, *freq: str*`)` → DataArray

Calculate the number of times low_data is below threshold while high_data is above threshold.

First, the threshold is transformed to the same standard_name and units as the input data, then the thresholding is performed, and finally, the number of occurrences is counted.

Parameters

- **low_data** (*xr.DataArray*) – Variable that must be under the threshold.
- **high_data** (*xr.DataArray*) – Variable that must be above the threshold.
- **threshold** (*str*) – Quantity.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray

`xclim.indices.generic.count_occurrences(`*data: DataArray*, *threshold: str*, *condition: str*, *freq: str*`)` → DataArray

Calculate the number of times some condition is met.

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, then this counts the number of times *data < threshold*. Finally, count the number of occurrences when condition is met.

Parameters

- **data** (*xr.DataArray*)
- **threshold** (*str*) – Quantity.
- **condition** (*{">", "<", ">=", "<=", "==", "!="}*) – Operator.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

`xclim.indices.generic.default_freq(`***indexer*`)` → str

Return the default frequency.

`xclim.indices.generic.degree_days(`*tas: DataArray*, *thresh: str*, *condition: str*`)` → DataArray

Calculate the degree days below/above the temperature threshold.

Parameters

- **tas** (*xr.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – The temperature threshold.
- **condition** (*{"<", ">"}*) – Operator.

Returns
xarray.DataArray

`xclim.indices.generic.diurnal_temperature_range(`*low_data: DataArray*, *high_data: DataArray*, *reducer: str*, *freq: str*`)` → DataArray

Calculate the diurnal temperature range and reduce according to a statistic.

Parameters

- **low_data** (*xr.DataArray*) – The lowest daily temperature (tasmin).
- **high_data** (*xr.DataArray*) – The highest daily temperature (tasmax).
- **reducer** (*{'max', 'min', 'mean', 'sum'}*) – Reducer.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

`xclim.indices.generic.domain_count(`*da: DataArray*, *low: float*, *high: float*, *freq: str*`)` → DataArray

Count number of days where value is within low and high thresholds.

A value is counted if it is larger than *low*, and smaller or equal to *high*, i.e. in *]low, high]*.

Parameters

- **da** (*xr.DataArray*) – Input data.
- **low** (*float*) – Minimum threshold value.

- **high** (*float*) – Maximum threshold value.
- **freq** (*str*) – Resampling frequency defining the periods defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling.

Returns
xr.DataArray – The number of days where value is within [low, high] for each period.

`xclim.indices.generic.doymax(`*da: DataArray*`)` → DataArray

Return the day of year of the maximum value.

`xclim.indices.generic.doymin(`*da: DataArray*`)` → DataArray

Return the day of year of the minimum value.

`xclim.indices.generic.get_daily_events(`*da: DataArray*, *da_value: float*, *operator: str*`)` → DataArray

Return a 0/1 mask when a condition is True or False.

Parameters

- **da** (*xr.DataArray*)
- **da_value** (*float*)
- **operator** (*{">", "<", ">=", "<=", "gt", "lt", "ge", "le"}*) – Logical operator {>, <, >=, <=, gt, lt, ge, le}. e.g. arr > thresh.

Notes

the function returns::

- 1 where operator(da, da_value) is True
- 0 where operator(da, da_value) is False
- nan where da is nan

Returns
xr.DataArray

`xclim.indices.generic.get_op(`*op: str*`)`

Get python's comparing function according to its name of representation.

Accepted op string are keys and values of xclim.indices.generic.binary_ops.

`xclim.indices.generic.interday_diurnal_temperature_range(`*low_data: DataArray*, *high_data: DataArray*, *freq: str*`)` → DataArray

Calculate the average absolute day-to-day difference in diurnal temperature range.

Parameters

- **low_data** (*xr.DataArray*) – The lowest daily temperature (tasmin).
- **high_data** (*xr.DataArray*) – The highest daily temperature (tasmax).
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

`xclim.indices.generic.last_occurrence(`*data: DataArray*, *threshold: str*, *condition: str*, *freq: str*`)` → DataArray

Calculate the last time some condition is met.

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Finally, locate the last occurrence when condition is met.

Parameters

- **data** (*xr.DataArray*)
- **threshold** (*str*) – Quantity
- **condition** (*{">", "<", ">=", "<=", "==", "!="}*) – Operator
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray

`xclim.indices.generic.select_resample_op(`*da: DataArray*, *op: str*, *freq: str = 'YS'*, ***indexer*`)` → DataArray

Apply operation over each period that is part of the index selection.

Parameters

- **da** (*xr.DataArray*) – Input data.
- **op** (*str {'min', 'max', 'mean', 'std', 'var', 'count', 'sum', 'argmax', 'argmin'} or func*) – Reduce operation. Can either be a DataArray method or a function that can be applied to a DataArray.
- **freq** (*str*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling.
- **indexer** (*{dim: indexer, }, optional*) – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered.

Returns

xarray.DataArray – The maximum value for each period.

`xclim.indices.generic.statistics(`*data: DataArray*, *reducer: str*, *freq: str*`)` → DataArray

Calculate a simple statistic of the data.

Parameters

- **data** (*xr.DataArray*)
- **reducer** (*{'max', 'min', 'mean', 'sum'}*) – Reducer.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray

`xclim.indices.generic.temperature_sum(`*data: DataArray*, *threshold: str*, *condition: str*, *freq: str*`)` → DataArray

Calculate the temperature sum above/below a threshold.

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold.

Finally, the sum is calculated for those data values that fulfill the condition after subtraction of the threshold value. If the sum is for values below the threshold the result is multiplied by -1.

Parameters

- **data** (*xr.DataArray*)
- **threshold** (*str*) – Quantity
- **condition** (*{">", "<", ">=", "<=", "==", "!="}*) – Operator
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

`xclim.indices.generic.threshold_count(`*da: xr.DataArray*, *op: str*, *thresh: float | int | xr.DataArray*, *freq: str*`)` → xr.DataArray

Count number of days where value is above or below threshold.

Parameters

- **da** (*xr.DataArray*) – Input data.
- **op** (*{">", "<", ">=", "<=", "gt", "lt", "ge", "le"}*) – Logical operator {>, <, >=, <=, gt, lt, ge, le }. e.g. arr > thresh.
- **thresh** (*Union[float, int]*) – Threshold value.
- **freq** (*str*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling.

Returns
xr.DataArray – The number of days meeting the constraints for each period.

`xclim.indices.generic.thresholded_statistics(`*data: DataArray*, *threshold: str*, *condition: str*, *reducer: str*, *freq: str*`)` → DataArray

Calculate a simple statistic of the data for which some condition is met.

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Finally, the statistic is calculated for those data values that fulfill the condition.

Parameters

- **data** (*xr.DataArray*)
- **threshold** (*str*) – Quantity.
- **condition** (*{">", "<", ">=", "<=", "==", "!="}*) – Operator
- **reducer** (*{'max', 'min', 'mean', 'sum'}*) – Reducer.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

Helper functions submodule

Functions that encapsulate some geophysical logic but could be shared by many indices.

`xclim.indices.helpers.cosine_of_solar_zenith_angle(`*declination: DataArray*, *lat: DataArray*, *lon: Optional[DataArray] = None*, *time_correction: Optional[DataArray] = None*, *hours: Optional[DataArray] = None*, *interval: Optional[int] = None*, *stat: str = 'integral'*`)` → DataArray

Cosine of the solar zenith angle.

The solar zenith angle is the angle between a vertical line (perpendicular to the ground) and the sun rays. This function computes a daily statistic of its cosine : its integral from sunrise to sunset or the average over the same period. Based on [Kalogirou14]. In addition it computes instantaneuos values of tis cosine. Based on [Napoli20].

Parameters

- **declination** (*xr.DataArray*) – Solar declination. See `solar_declination()`.
- **lat** (*xr.DataArray*) – Latitude.
- **lon** (*xr.DataArray, optional*) – Longitude This is necessary if stat is "instant", "interval" or "sunlit".
- **time_correction** (*xr.DataArray, optional*) – Time correction for solar angle. See `time_correction_for_solar_angle()` This is necessary if stat is "instant".
- **hours** (*xr.DataArray, optional*) – Watch time hours. This is necessary if stat is "instant", "interval" or "sunlit".
- **interval** (*int, optional*) – Time interal between two time steps in hours This is necessary if stat is "interval" or "sunlit".
- **stat** (*{'integral', 'average', 'instant', 'interval', 'sunlit'}*) – Which daily statistic to return. If "integral", this returns the integral of the cosine of the zenith angle from sunrise to sunset. If "average", the integral is divided by the "duration" from sunrise to sunset. If "instant", this returns the instantaneous cosine of the zenith angle. If "interval", this returns the cosine of the zenith angle during each interval. If "sunlit", this returns the cosine of the zenith angle during the sunlit period of each interval.

Returns

Cosine of the solar zenith angle, [rad] or [dimensionless] – If stat is "integral", dimensions can be said to be "time" as the integral is on the hour angle. For seconds, multiply by the number of seconds in a comple day cycle (24*60*60) and divide by 2.

Notes

This code was inspired by the *thermofeel* and *PyWBGT* package.

References

Kalogirou, S. A. (2014). Chapter 2 — Environmental Characteristics. In S. A. Kalogirou (Ed.), Solar Energy Engineering (Second Edition) (pp. 51–123). Academic Press. https://doi.org/10.1016/B978-0-12-397270-5.00002-9 Di Napoli, C., Hogan, R.J. & Pappenberger, F. Mean radiant temperature from global-scale numerical weather prediction models. Int J Biometeorol 64, 1233–1245 (2020). https://doi.org/10.1007/s00484-020-01900-5

`xclim.indices.helpers.day_lengths(`*dates: DataArray*, *lat: DataArray*, *method: str = 'spencer'*`)` → DataArray

Day-lengths according to latitude and day of year.

See *`solar_declination()`* for the approximation used to compute the solar declination angle. Based on [Kalogirou14].

Parameters

- **dates** (*xr.DataArray*)
- **lat** (*xarray.DataArray*) – Latitude coordinate.
- **method** (*{'spencer', 'simple'}*) – Which approximation to use when computing the solar declination angle. See *`solar_declination()`*.

Returns

xarray.DataArray, [hours] – Day-lengths in hours per individual day.

References

Kalogirou, S. A. (2014). Chapter 2 — Environmental Characteristics. In S. A. Kalogirou (Ed.), Solar Energy Engineering (Second Edition) (pp. 51–123). Academic Press. https://doi.org/10.1016/B978-0-12-397270-5.00002-9

`xclim.indices.helpers.distance_from_sun(`*dates: xr.DaraArray*`)` → xr.DataArray

Sun-earth distance.

The distance from sun to earth in astronomical units.

Parameters

dates (*xr.DataArray*) – Series of dates and time of days

Returns

xr.DataArray, [astronomical units] – Sun-earth distance

References

U.S. Naval Observatory:Astronomical Almanac. Washington, D.C.: U.S. Government Printing Office (1985).

`xclim.indices.helpers.eccentricity_correction_factor(`*day_ angle: DataArray*, *method='spencer'*`)`

Eccentricity correction factor of the Earth's orbit.

The squared ratio of the mean distance Earth-Sun to the distance at a specific moment. As approximated by [Spencer1971].

Parameters

- **day_angle** (*xr.DataArray*) – Assuming the earth makes a full circle in a year, this is the angle covered from the beginning of the year up to that timestep. Also called the "julian day fraction". See `datetime_to_decimal_year()`.
- **method** – Which approximation to use. The default ("spencer") uses the first five terms of the fourier series of the eccentrencity, while "simple" approximates with only the first two.

Returns

Eccentricity correction factor, [dimensionless]

References

Spencer JW (1971) Fourier series representation of the position of the sun. Search 2(5):172

`xclim.indices.helpers.extraterrestrial_solar_radiation(`*times: DataArray, lat: DataArray, solar_constant: str = '1361 W m-2', method='spencer'*`)` → DataArray

Extraterrestrial solar radiation.

This is the daily energy received on a surface parallel to the ground at the mean distance of the earth to the sun. It neglects the effect of the atmosphere. Computation is based on [Kalogirou14] and the default solar constant is taken from [Matthes17].

Parameters

- **times** (*xr.DataArray*) – Daily datetime data. This function makes no sense with data of other frequency.
- **lat** (*xr.DataArray*) – Latitude.
- **solar_constant** (*str*) – The solar constant, the energy received on earth from the sun per surface per time.
- **method** (*{'spencer', 'simple'}*) – Which method to use when computing the solar declination and the eccentricity correction factor. See `solar_declination()` and `eccentricity_correction_factor()`.

Returns

Extraterrestrial solar radiation, [J m-2 d-1]

References

`xclim.indices.helpers.solar_declination(`*day_angle: DataArray, method='spencer'*`)` → DataArray

Solar declination.

The angle between the sun rays and the earth's equator, in radians, as approximated by [Spencer1971] or assuming the orbit is a cirle.

Parameters

- **day_angle** (*xr.DataArray*) – Assuming the earth makes a full circle in a year, this is the angle covered from the beginning of the year up to that timestep. Also called the "julian day fraction". See `datetime_to_decimal_year()`.
- **method** (*{'spencer', 'simple'}*) – Which approximation to use. The default ("spencer") uses the first 7 terms of the Fourier series representing the observed declination, while "simple" assumes the orbit is a circle with a fixed obliquity and

that the solstice/equinox happen at fixed angles on the orbit (the exact calendar date changes for leap years).

Returns
Solar declination angle, [rad]

References

`xclim.indices.helpers.time_correction_for_solar_angle(`*day_angle: DataArray*`)` → DataArray

Time correction for solar angle.

Every 1° of angular rotation on earth is equal to 4 minutes of time. The time correction helpsis needed to correct local watch time to solar time.

Parameters
day_angle (*xr.DataArray*) – Assuming the earth makes a full circle in a year, this is the angle covered from the beginning of the year up to that timestep. Also called the "julian day fraction". See `datetime_to_decimal_year()`.

Returns
Time correction of solar angle, [rad]

References

Di Napoli, C., Hogan, R.J. & Pappenberger, F. Mean radiant temperature from global-scale numerical weather prediction models. Int J Biometeorol 64, 1233–1245 (2020). https://doi.org/10.1007/s00484-020-01900-5

Run length algorithms submodule

Computation of statistics on runs of True values in boolean arrays.

`xclim.indices.run_length.first_run(`*da: xr.DataArray, window: int, dim: str = 'time', coord: str | bool | None = False, ufunc_1dim: str | bool = 'from_context'*`)` → xr.DataArray

Return the index of the first item of the first run of at least a given length.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values. When equal to 1, an optimized version of the algorithm is used.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **coord** (*Optional[str]*) – If not False, the function returns values along *dim* instead of indexes. If *dim* has a datetime dtype, *coord* can also be a str of the name of the DateTimeAccessor object to use (ex: 'dayofyear').
- **ufunc_1dim** (*Union[str, bool]*) – Use the 1d 'ufunc' version of this function : default (auto) will attempt to select optimal usage based on number of data points. Using 1D_ufunc=True is typically more efficient for DataArray with a small number of grid points. Ignored when *window=1*. It can be modified globally through the "run_length_ufunc" global option.

Returns
xr.DataArray – Index (or coordinate if *coord* is not False) of first item in first valid run. Returns np.nan if there are no valid runs.

`xclim.indices.run_length.first_run_1d(`*arr: Sequence[int | float]*, *window: int*`)` → int

Return the index of the first item of a run of at least a given length.

Parameters

- **arr** (*Sequence[Union[int, float]]*) – Input array.
- **window** (*int*) – Minimum duration of consecutive run to accumulate values.

Returns
int – Index of first item in first valid run. Returns np.nan if there are no valid runs.

`xclim.indices.run_length.first_run_after_date(`*da: xr.DataArray*, *window: int*, *date: DayOfYearStr | None = '07-01'*, *dim: str = 'time'*, *coord: bool | str | None = 'dayofyear'*`)` → xr.DataArray

Return the index of the first item of the first run after a given date.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values.
- **date** (*DayOfYearStr*) – The date after which to look for the run.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **coord** (*Optional[Union[bool, str]]*) – If not False, the function returns values along *dim* instead of indexes. If *dim* has a datetime dtype, *coord* can also be a str of the name of the DateTimeAccessor object to use (ex: 'dayofyear').

Returns
xr.DataArray – Index (or coordinate if *coord* is not False) of first item in the first valid run. Returns np.nan if there are no valid runs.

`xclim.indices.run_length.first_run_ufunc(`*x: xr.DataArray | Sequence[bool]*, *window: int*, *dim: str*`)` → xr.DataArray

Dask-parallel version of first_run_1d, ie: the first entry in array of consecutive true values.

Parameters

- **x** (*Union[xr.DataArray, Sequence[bool]]*) – Input array (bool).
- **window** (*int*) – Minimum run length.
- **dim** (*str*) – The dimension along which the runs are found.

Returns
xr.DataArray – A function operating along the time dimension of a dask-array.

`xclim.indices.run_length.index_of_date(`*time: xr.DataArray*, *date: DateStr | DayOfYearStr | None*, *max_idxs: int | None = None*, *default: int = 0*`)` → np.ndarray

Get the index of a date in a time array.

Parameters

- **time** (*xr.DataArray*) – An array of datetime values, any calendar.

- **date** (*DayOfYearStr or DateStr, optional*) – A string in the "yyyy-mm-dd" or "mm-dd" format. If None, returns default.
- **max_idxs** (*int, optional*) – Maximum number of returned indexes.
- **default** (*int*) – Index to return if date is None.

Raises
`ValueError` – If there are most instances of *date* in *time* than *max_idxs*.

Returns
numpy.ndarray – 1D array of integers, indexes of *date* in *time*.

`xclim.indices.run_length.keep_longest_run(`*da: DataArray*, *dim: str = 'time'*`)` → DataArray

Keep the longest run along a dimension.

Parameters

- **da** (*xr.DataArray*) – Boolean array.
- **dim** (*str*) – Dimension along which to check for the longest run.

Returns
xr.DataArray – Boolean array similar to da but with only one run, the (first) longest.

`xclim.indices.run_length.last_run(`*da: xr.DataArray*, *window: int*, *dim: str = 'time'*, *coord: str | bool | None = False*, *ufunc_1dim: str | bool = 'from_context'*`)` → xr.DataArray

Return the index of the last item of the last run of at least a given length.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values. When equal to 1, an optimized version of the algorithm is used.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **coord** (*Optional[str]*) – If not False, the function returns values along *dim* instead of indexes. If *dim* has a datetime dtype, *coord* can also be a str of the name of the DateTimeAccessor object to use (ex: 'dayofyear').
- **ufunc_1dim** (*Union[str, bool]*) – Use the 1d 'ufunc' version of this function : default (auto) will attempt to select optimal usage based on number of data points. Using *1D_ufunc=True* is typically more efficient for a DataArray with a small number of grid points. Ignored when *window=1*. It can be modified globally through the "run_length_ufunc" global option.

Returns
xr.DataArray – Index (or coordinate if *coord* is not False) of last item in last valid run. Returns np.nan if there are no valid runs.

`xclim.indices.run_length.last_run_before_date(`*da: xr.DataArray*, *window: int*, *date: DayOfYearStr = '07-01'*, *dim: str = 'time'*, *coord: bool | str | None = 'dayofyear'*`)` → xr.DataArray

Return the index of the last item of the last run before a given date.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values.

- **date** (*DayOfYearStr*) – The date before which to look for the last event.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **coord** (*Optional[Union[bool, str]]*) – If not False, the function returns values along *dim* instead of indexes. If *dim* has a datetime dtype, *coord* can also be a str of the name of the DateTimeAccessor object to use (ex: 'dayofyear').

Returns
xr.DataArray – Index (or coordinate if *coord* is not False) of last item in last valid run. Returns np.nan if there are no valid runs.

`xclim.indices.run_length.lazy_indexing(`*da: xr.DataArray*, *index: xr.DataArray*, *dim: str | None = None*`)` → xr.DataArray

Get values of *da* at indices *index* in a NaN-aware and lazy manner.

Two case

Parameters

- **da** (*xr.DataArray*) – Input array. If not 1D, *dim* must be given and must not appear in index.
- **index** (*xr.DataArray*) – N-d integer indices, if da is not 1D, all dimensions of index must be in da
- **dim** (*str, optional*) – Dimension along which to index, unused if *da* is 1D, should not be present in *index*.

Returns
xr.DataArray – Values of *da* at indices *index*.

`xclim.indices.run_length.longest_run(`*da: xr.DataArray*, *dim: str = 'time'*, *ufunc_1dim: str | bool = 'from_context'*, *index: str = 'first'*`)` → xr.DataArray

Return the length of the longest consecutive run of True values.

Parameters

- **da** (*xr.DataArray*) – N-dimensional array (boolean)
- **dim** (*str*) – Dimension along which to calculate consecutive run; Default: 'time'.
- **ufunc_1dim** (*Union[str, bool]*) – Use the 1d 'ufunc' version of this function : default (auto) will attempt to select optimal usage based on number of data points. Using 1D_ufunc=True is typically more efficient for DataArray with a small number of grid points. It can be modified globally through the "run_length_ufunc" global option.
- **index** (*{'first', 'last'}*) – If 'first', the run length is indexed with the first element in the run. If 'last', with the last element in the run.

Returns
xr.DataArray – Length of the longest run of True values along dimension (int).

`xclim.indices.run_length.npts_opt = 9000`

Arrays with less than this number of data points per slice will trigger the use of the ufunc version of run lengths algorithms.

`xclim.indices.run_length.rle(`*da: DataArray*, *dim: str = 'time'*, *index: str = 'first'*`)` → DataArray

Generate basic run length function.

Parameters

- **da** (*xr.DataArray*) – Input array.
- **dim** (*str*) – Dimension name.
- **index** (*{'first', 'last'}*) – If 'first' (default), the run length is indexed with the first element in the run. If 'last', with the last element in the run.

Returns
xr.DataArray – Values are 0 where da is False (out of runs).

`xclim.indices.run_length.rle_1d`(*arr: int | float | bool | Sequence[int | float | bool]*) → tuple[np.array, np.array, np.array]

Return the length, starting position and value of consecutive identical values.

Parameters
arr (*Sequence[Union[int, float, bool]]*) – Array of values to be parsed.

Returns

- **values** (*np.array*) – The values taken by arr over each run.
- **run lengths** (*np.array*) – The length of each run.
- **start position** (*np.array*) – The starting index of each run.

Examples

```
>>> from xclim.indices.run_length import rle_1d
>>> a = [1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3]
>>> rle_1d(a)
(array([1, 2, 3]), array([2, 4, 6]), array([0, 2, 6]))
```

`xclim.indices.run_length.rle_statistics`(*da: xr.DataArray, reducer: str = 'max', window: int = 1, dim: str = 'time', ufunc_1dim: str | bool = 'from_context', index: str = 'first'*) → xr.DataArray

Return the length of consecutive run of True values, according to a reducing operator.

Parameters

- **da** (*xr.DataArray*) – N-dimensional array (boolean).
- **reducer** (*str*) – Name of the reducing function.
- **window** (*int*) – Minimal length of consecutive runs to be included in the statistics.
- **dim** (*str*) – Dimension along which to calculate consecutive run; Default: 'time'.
- **ufunc_1dim** (*Union[str, bool]*) – Use the 1d 'ufunc' version of this function : default (auto) will attempt to select optimal usage based on number of data points. Using 1D_ufunc=True is typically more efficient for DataArray with a small number of grid points. It can be modified globally through the "run_length_ufunc" global option.
- **index** (*{'first', 'last'}*) – If 'first' (default), the run length is indexed with the first element in the run. If 'last', with the last element in the run.

Returns
xr.DataArray – Length of runs of True values along dimension, according to the reducing function (float) If there are no runs (but the data is valid), returns 0.

`xclim.indices.run_length.run_bounds(`*mask: xr.DataArray*, *dim: str = 'time'*, *coord: bool | str | None = True*`)`

Return the start and end dates of boolean runs along a dimension.

Parameters

- **mask** (*xr.DataArray*) – Boolean array.
- **dim** (*str*) – Dimension along which to look for runs.
- **coord** (*bool or str*) – If True, return values of the coordinate, if a string, returns values from *dim.dt.<coord>*. if False, return indexes.

Returns

xr.DataArray – With `dim` reduced to "events" and "bounds". The events dim is as long as needed, padded with NaN or NaT.

`xclim.indices.run_length.run_end_after_date(`*da: xr.DataArray*, *window: int*, *date: DayOfYearStr = '07-01'*, *dim: str = 'time'*, *coord: bool | str | None = 'dayofyear'*`)` → xr.DataArray

Return the index of the first item after the end of a run after a given date.

The run must begin before the date.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values.
- **date** (*str*) – The date after which to look for the end of a run.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **coord** (*Optional[Union[bool, str]]*) – If not False, the function returns values along *dim* instead of indexes. If *dim* has a datetime dtype, *coord* can also be a str of the name of the DateTimeAccessor object to use (ex: 'dayofyear').

Returns

xr.DataArray – Index (or coordinate if *coord* is not False) of last item in last valid run. Returns np.nan if there are no valid runs.

`xclim.indices.run_length.season(`*da: xr.DataArray*, *window: int*, *date: DayOfYearStr | None = None*, *dim: str = 'time'*, *coord: str | bool | None = False*`)` → xr.Dataset

Return the bounds of a season (along dim).

A "season" is a run of True values that may include breaks under a given length (*window*). The start is computed as the first run of *window* True values, then end as the first subsequent run of *window* False values. If a date is passed, it must be included in the season.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive values to start and end the season.
- **date** (*DayOfYearStr, optional*) – The date (in MM-DD format) that a run must include to be considered valid.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **coord** (*Optional[str]*) – If not False, the function returns values along *dim* instead of indexes. If *dim* has a datetime dtype, *coord* can also be a str of the name of the DateTimeAccessor object to use (ex: 'dayofyear').

Returns

xr.Dataset – "dim" is reduced to "season_bnds" with 2 elements : season start and season end, both indices of da[dim].

Notes

The run can include holes of False or NaN values, so long as they do not exceed the window size.

If a date is given, the season start and end are forced to be on each side of this date. This means that even if the "real" season has been over for a long time, this is the date used in the length calculation. Example : Length of the "warm season", where T > 25°C, with date = 1st August. Let's say the temperature is over 25 for all june, but july and august have very cold temperatures. Instead of returning 30 days (june), the function will return 61 days (july + june).

`xclim.indices.run_length.season_length(`*da: xr.DataArray*, *window: int*, *date: DayOfYearStr | None = None*, *dim: str = 'time'*`)` → xr.DataArray

Return the length of the longest semi-consecutive run of True values (optionally including a given date).

A "season" is a run of True values that may include breaks under a given length (*window*). The start is computed as the first run of *window* True values, then end as the first subsequent run of *window* False values. If a date is passed, it must be included in the season.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive values to start and end the season.
- **date** (*DayOfYearStr, optional*) – The date (in MM-DD format) that a run must include to be considered valid.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').

Returns

xr.DataArray – Length of the longest run of True values along a given dimension (inclusive of a given date) without breaks longer than a given length.

Notes

The run can include holes of False or NaN values, so long as they do not exceed the window size.

If a date is given, the season end is forced to be later or equal to this date. This means that even if the "real" season has been over for a long time, this is the date used in the length calculation. Example : Length of the "warm season", where T > 25°C, with date = 1st August. Let's say the temperature is over 25 for all june, but july and august have very cold temperatures. Instead of returning 30 days (june), the function will return 61 days (july + june).

`xclim.indices.run_length.statistics_run_1d(`*arr: Sequence[bool]*, *reducer: str*, *window: int = 1*`)` → int

Return statistics on lengths of run of identical values.

Parameters

- **arr** (*Sequence[bool]*) – Input array (bool)
- **reducer** (*{'mean', 'sum', 'min', 'max', 'std'}*) – Reducing function name.
- **window** (*int*) – Minimal length of runs to be included in the statistics

Returns
int – Statistics on length of runs.

`xclim.indices.run_length.statistics_run_ufunc(`*x: xr.DataArray | Sequence[bool], reducer: str, window: int = 1, dim: str = 'time'*`)` → xr.DataArray

Dask-parallel version of statistics_run_1d, ie: the {reducer} number of consecutive true values in array.

Parameters

- **x** (*Sequence[bool]*) – Input array (bool)
- **reducer** (*{'min', 'max', 'mean', 'sum', 'std'}*) – Reducing function name.
- **window** (*int*) – Minimal length of runs.
- **dim** (*str*) – The dimension along which the runs are found.

Returns
xr.DataArray – A function operating along the time dimension of a dask-array.

`xclim.indices.run_length.suspicious_run(`*arr: xr.DataArray, dim: str = 'time', window: int = 10, op: str = '>', thresh: float | None = None*`)` → xr.DataArray

Return True where the array contains has runs of identical values, vectorized version.

In opposition to other run length functions, here the output has the same shape as the input.

Parameters

- **arr** (*xr.DataArray*) – Array of values to be parsed.
- **dim** (*str*) – Dimension along which to check for runs (default: "time").
- **window** (*int*) – Minimum run length
- **thresh** (*float, optional*) – Threshold above which values are checked for identical values.
- **op** (*{">", ">=", "==", "<", "<=", "eq", "gt", "lt", "gteq", "lteq"}*) – Operator for threshold comparison, defaults to ">".

Returns
xarray.DataArray

`xclim.indices.run_length.suspicious_run_1d(`*arr: np.ndarray, window: int = 10, op: str = '>', thresh: float | None = None*`)` → np.ndarray

Return True where the array contains a run of identical values.

Parameters

- **arr** (*numpy.ndarray*) – Array of values to be parsed.
- **window** (*int*) – Minimum run length
- **op** (*{">", ">=", "==", "<", "<=", "eq", "gt", "lt", "gteq", "lteq"}, optional*) – Operator for threshold comparison. Defaults to ">".
- **thresh** (*float, optional*) – Threshold above which values are checked for identical values.

Returns
numpy.ndarray – Whether or not the data points are part of a run of identical values.

`xclim.indices.run_length.use_ufunc(`*ufunc_1dim: bool | str*, *da: xr.DataArray*, *dim: str = 'time'*, *index: str = 'first'*`)` → bool

Return whether the ufunc version of run length algorithms should be used with this DataArray or not.

If ufunc_1dim is 'from_context', the parameter is read from xclim's global (or context) options. If it is 'auto', this returns False for dask-backed array and for arrays with more than `npts_opt` points per slice along *dim*.

Parameters

- **ufunc_1dim** (*{'from_context', 'auto', True, False}*) – The method for handling the ufunc parameters.
- **da** (*xr.DataArray*) – Input array.
- **dim** (*str*) – The dimension along which to find runs.
- **index** (*{'first', 'last'}*) – If 'first' (default), the run length is indexed with the first element in the run. If 'last', with the last element in the run.

Returns

bool – If ufunc_1dim is "auto", returns True if the array is on dask or too large. Otherwise, returns ufunc_1dim.

`xclim.indices.run_length.windowed_run_count(`*da: xr.DataArray*, *window: int*, *dim: str = 'time'*, *ufunc_1dim: str | bool = 'from_context'*, *index: str = 'first'*`)` → xr.DataArray

Return the number of consecutive true values in array for runs at least as long as given duration.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum run length. When equal to 1, an optimized version of the algorithm is used.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **ufunc_1dim** (*Union[str, bool]*) – Use the 1d 'ufunc' version of this function : default (auto) will attempt to select optimal usage based on number of data points. Using 1D_ufunc=True is typically more efficient for DataArray with a small number of grid points. Ignored when *window=1*. It can be modified globally through the "run_length_ufunc" global option.
- **index** (*{'first', 'last'}*) – If 'first', the run length is indexed with the first element in the run. If 'last', with the last element in the run.

Returns

xr.DataArray – Total number of *True* values part of a consecutive runs of at least *window* long.

`xclim.indices.run_length.windowed_run_count_1d(`*arr: Sequence[bool]*, *window: int*`)` → int

Return the number of consecutive true values in array for runs at least as long as given duration.

Parameters

- **arr** (*Sequence[bool]*) – Input array (bool).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values.

Returns

int – Total number of true values part of a consecutive run at least *window* long.

`xclim.indices.run_length.windowed_run_count_ufunc(`*x: xr.DataArray | Sequence[bool]*, *window: int*, *dim: str*`)` → xr.DataArray

Dask-parallel version of windowed_run_count_1d, ie: the number of consecutive true values in array for runs at least as long as given duration.

Parameters

- **x** (*Sequence[bool]*) – Input array (bool).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values.
- **dim** (*str*) – Dimension along which to calculate windowed run.

Returns

xr.DataArray – A function operating along the time dimension of a dask-array.

`xclim.indices.run_length.windowed_run_events(`*da: xr.DataArray*, *window: int*, *dim: str = 'time'*, *ufunc_1dim: str | bool = 'from_context'*, *index: str = 'first'*`)` → xr.DataArray

Return the number of runs of a minimum length.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum run length. When equal to 1, an optimized version of the algorithm is used.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **ufunc_1dim** (*Union[str, bool]*) – Use the 1d 'ufunc' version of this function : default (auto) will attempt to select optimal usage based on number of data points. Using 1D_ufunc=True is typically more efficient for DataArray with a small number of grid points. Ignored when *window=1*. It can be modified globally through the "run_length_ufunc" global option.
- **index** (*{'first', 'last'}*) – If 'first', the run length is indexed with the first element in the run. If 'last', with the last element in the run.

Returns

xr.DataArray – Number of distinct runs of a minimum length (int).

`xclim.indices.run_length.windowed_run_events_1d(`*arr: Sequence[bool]*, *window: int*`)` → DataArray

Return the number of runs of a minimum length.

Parameters

- **arr** (*Sequence[bool]*) – Input array (bool).
- **window** (*int*) – Minimum run length.

Returns

xr.DataArray – Number of distinct runs of a minimum length.

`xclim.indices.run_length.windowed_run_events_ufunc(`*x: xr.DataArray | Sequence[bool]*, *window: int*, *dim: str*`)` → xr.DataArray

Dask-parallel version of windowed_run_events_1d, ie: the number of runs at least as long as given duration.

Parameters

- **x** (*Sequence[bool]*) – Input array (bool).
- **window** (*int*) – Minimum run length.

- **dim** (*str*) – Dimension along which to calculate windowed run.

Returns
xr.DataArray – A function operating along the time dimension of a dask-array.

14.3 Ensembles module

Ensemble tools.

This submodule defines some useful methods for dealing with ensembles of climate simulations. In xclim, an "ensemble" is a *Dataset* or a *DataArray* where multiple climate realizations or models are concatenated along the *realization* dimension.

`xclim.ensembles.create_ensemble(`*datasets: list[xr.Dataset | xr.DataArray | Path | str | list[Path | str]] | str*, *mf_flag: bool = False*, *resample_freq: str | None = None*, *calendar: str = 'default'*, ***xr_kwargs*`)` → xr.Dataset

Create an xarray dataset of an ensemble of climate simulation from a list of netcdf files.

Input data is concatenated along a newly created data dimension ('realization'). Returns an xarray dataset object containing input data from the list of netcdf files concatenated along a new dimension (name:'realization'). In the case where input files have unequal time dimensions, the output ensemble Dataset is created for maximum time-step interval of all input files. Before concatenation, datasets not covering the entire time span have their data padded with NaN values. Dataset and variable attributes of the first dataset are copied to the resulting dataset.

Parameters

- **datasets** (*List[Union[xr.Dataset, Path, str, List[Path, str]]] or str*) – List of netcdf file paths or xarray Dataset/DataArray objects . If mf_flag is True, ncfiles should be a list of lists where each sublist contains input .nc files of an xarray multifile Dataset. If DataArray object are passed, they should have a name in order to be transformed into Datasets. If a string is passed, it is assumed to be a glob pattern for finding datasets.
- **mf_flag** (*bool*) – If True, climate simulations are treated as xarray multifile Datasets before concatenation. Only applicable when "datasets" is a sequence of file paths.
- **resample_freq** (*Optional[str]*) – If the members of the ensemble have the same frequency but not the same offset, they cannot be properly aligned. If resample_freq is set, the time coordinate of each members will be modified to fit this frequency.
- **calendar** (*str*) – The calendar of the time coordinate of the ensemble. For conversions involving '360_day', the align_on='date' option is used. See *xclim.core.calendar.convert_calendar*. 'default' is the standard calendar using np.datetime64 objects.
- **xr_kwargs** – Any keyword arguments to be given to *xr.open_dataset* when opening the files (or to *xr.open_mfdataset* if mf_flag is True)

Returns
xr.Dataset – Dataset containing concatenated data from all input files.

Notes

Input netcdf files require equal spatial dimension size (e.g. lon, lat dimensions). If input data contains multiple cftime calendar types they must be at monthly or coarser frequency.

Examples

```
>>> from xclim.ensembles import create_ensemble
>>> ens = create_ensemble(temperature_datasets)
```

Using multifile datasets, through glob patterns. Simulation 1 is a list of .nc files (e.g. separated by time):

```
>>> datasets = glob.glob("/dir/*.nc")
```

Simulation 2 is also a list of .nc files:

```
>>> datasets.append(glob.glob("/dir2/*.nc"))
>>> ens = create_ensemble(datasets, mf_flag=True)
```

`xclim.ensembles.ensemble_mean_std_max_min`(*ens: Dataset*) → Dataset

Calculate ensemble statistics between a results from an ensemble of climate simulations.

Returns an xarray Dataset containing ensemble mean, standard-deviation, minimum and maximum for input climate simulations.

Parameters

ens (*xr.Dataset*) – Ensemble dataset (see xclim.ensembles.create_ensemble).

Returns

xr.Dataset – Dataset with data variables of ensemble statistics.

Examples

```
>>> from xclim.ensembles import create_ensemble, ensemble_mean_std_max_min
```

Create the ensemble dataset:

```
>>> ens = create_ensemble(temperature_datasets)
```

Calculate ensemble statistics:

```
>>> ens_mean_std = ensemble_mean_std_max_min(ens)
```

`xclim.ensembles.ensemble_percentiles`(*ens: xr.Dataset | xr.DataArray, values: Sequence[float] = [10, 50, 90], keep_chunk_size: bool | None = None, split: bool = True*) → xr.Dataset

Calculate ensemble statistics between a results from an ensemble of climate simulations.

Returns a Dataset containing ensemble percentiles for input climate simulations.

Parameters

- **ens** (*Union[xr.Dataset, xr.DataArray]*) – Ensemble dataset or dataarray (see xclim.ensembles.create_ensemble).

- **values** (*Tuple[int, int, int]*) – Percentile values to calculate. Default: (10, 50, 90).
- **keep_chunk_size** (*Optional[bool]*) – For ensembles using dask arrays, all chunks along the 'realization' axis are merged. If True, the dataset is rechunked along the dimension with the largest chunks, so that the chunks keep the same size (approx) If False, no shrinking is performed, resulting in much larger chunks If not defined, the function decides which is best
- **split** (*bool*) – Whether to split each percentile into a new variable of concatenate the ouput along a new "percentiles" dimension.

Returns

Union[xr.Dataset, xr.DataArray] – If split is True, same type as ens; dataset otherwise, containing data variable(s) of requested ensemble statistics

Examples

```
>>> from xclim.ensembles import create_ensemble, ensemble_percentiles
```

Create ensemble dataset:

```
>>> ens = create_ensemble(temperature_datasets)
```

Calculate default ensemble percentiles:

```
>>> ens_percs = ensemble_percentiles(ens)
```

Calculate non-default percentiles (25th and 75th)

```
>>> ens_percs = ensemble_percentiles(ens, values=(25, 50, 75))
```

If the original array has many small chunks, it might be more efficient to do:

```
>>> ens_percs = ensemble_percentiles(ens, keep_chunk_size=False)
```

14.3.1 Ensemble Reduction

Ensemble reduction is the process of selecting a subset of members from an ensemble in order to reduce the volume of computation needed while still covering a good portion of the simulated climate variability.

`xclim.ensembles.kkz_reduce_ensemble`(*data: DataArray*, *num_select: int*, ***, *dist_method: str = 'euclidean'*, *standardize: bool = True*, ***cdist_kwargs*) → list

Return a sample of ensemble members using KKZ selection.

The algorithm selects *num_select* ensemble members spanning the overall range of the ensemble. The selection is ordered, smaller groups are always subsets of larger ones for given criteria. The first selected member is the one nearest to the centroid of the ensemble, all subsequent members are selected in a way maximizing the phase-space coverage of the group. Algorithm taken from [CannonKKZ].

Parameters

- **data** (*xr.DataArray*) – Selection criteria data : 2-D xr.DataArray with dimensions 'realization' (N) and 'criteria' (P). These are the values used for clustering. Realizations represent the individual original ensemble members and criteria the variables/indicators used in the grouping algorithm.

- **num_select** (*int*) – The number of members to select.
- **dist_method** (*str*) – Any distance metric name accepted by *scipy.spatial.distance.cdist*.
- **standardize** (*bool*) – Whether to standardize the input before running the selection or not. Standardization consists in translation as to have a zero mean and scaling as to have a unit standard deviation.
- **cdist_kwargs** – All extra arguments are passed as-is to *scipy.spatial.distance.cdist*, see its docs for more information.

Returns

list – Selected model indices along the *realization* dimension.

References

`xclim.ensembles.kmeans_reduce_ensemble(`*data: xarray.DataArray*, ***, *method: dict = None*, *make_graph: bool = True*, *max_clusters: int | None = None*, *variable_weights: np.ndarray | None = None*, *model_weights: np.ndarray | None = None*, *sample_weights: np.ndarray | None = None*, *random_state: int | np.random.RandomState | None = None*`)` → tuple[list, np.ndarray, dict]

Return a sample of ensemble members using k-means clustering.

The algorithm attempts to reduce the total number of ensemble members while maintaining adequate coverage of the ensemble uncertainty in an N-dimensional data space. K-Means clustering is carried out on the input selection criteria data-array in order to group individual ensemble members into a reduced number of similar groups. Subsequently, a single representative simulation is retained from each group.

Parameters

- **data** (*xr.DataArray*) – Selecton criteria data : 2-D xr.DataArray with dimensions 'realization' (N) and 'criteria' (P). These are the values used for clustering. Realizations represent the individual original ensemble members and criteria the variables/indicators used in the grouping algorithm.
- **method** (*dict*) – Dictionary defining selection method and associated value when required. See Notes.
- **max_clusters** (*Optional[int]*) – Maximum number of members to include in the output ensemble selection. When using 'rsq_optimize' or 'rsq_cutoff' methods, limit the final selection to a maximum number even if method results indicate a higher value. Defaults to N.
- **variable_weights** (*Optional[np.ndarray]*) – An array of size P. This weighting can be used to influence of weight of the climate indices (criteria dimension) on the clustering itself.
- **model_weights** (*Optional[np.ndarray]*) – An array of size N. This weighting can be used to influence which realization is selected from within each cluster. This parameter has no influence on the clustering itself.
- **sample_weights** (*Optional[np.ndarray]*) – An array of size N. sklearn.cluster.KMeans() sample_weights parameter. This weighting can be used to influence of weight of simulations on the clustering itself. See: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

- **random_state** (*Optional[Union[int, np.random.RandomState]]*) – sklearn.cluster.KMeans() random_state parameter. Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. See: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
- **make_graph** (*bool*) – output a dictionary of input for displays a plot of R^2 vs. the number of clusters. Defaults to True if matplotlib is installed in runtime environment.

Notes

Parameters for method in call must follow these conventions:

rsq_optimize

Calculate coefficient of variation (R^2) of cluster results for n = 1 to N clusters and determine an optimal number of clusters that balances cost / benefit tradeoffs. This is the default setting. See supporting information S2 text in https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0152495

method={'rsq_optimize':None}

rsq_cutoff

Calculate Coefficient of variation (R^2) of cluster results for n = 1 to N clusters and determine the minimum numbers of clusters needed for $R^2 >$ val.

val : float between 0 and 1. R^2 value that must be exceeded by clustering results.

method={'rsq_cutoff': val}

n_clusters

Create a user determined number of clusters.

val : integer between 1 and N

method={'n_clusters': val}

Returns

- *list* – Selected model indexes (positions)
- *np.ndarray* – KMeans clustering results
- *dict* – Dictionary of input data for creating R^2 profile plot. 'None' when make_graph=False

References

Casajus et al. 2016. https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0152495

Examples

```
>>> import xclim
>>> from xclim.ensembles import create_ensemble, kmeans_reduce_ensemble
>>> from xclim.indices import hot_spell_frequency
```

Start with ensemble datasets for temperature:

```
>>> ensTas = create_ensemble(temperature_datasets)
```

Calculate selection criteria – Use annual climate change fields between 2071-2100 and 1981-2010 normals. First, average annual temperature:

```
>>> tg = xclim.atmos.tg_mean(tas=ensTas.tas)
>>> his_tg = tg.sel(time=slice("1990", "2019")).mean(dim="time")
>>> fut_tg = tg.sel(time=slice("2020", "2050")).mean(dim="time")
>>> dtg = fut_tg - his_tg
```

Then, Hotspell frequency as second indicator:

```
>>> hs = hot_spell_frequency(tasmax=ensTas.tas, window=2, thresh_tasmax="10 degC")
>>> his_hs = hs.sel(time=slice("1990", "2019")).mean(dim="time")
>>> fut_hs = hs.sel(time=slice("2020", "2050")).mean(dim="time")
>>> dhs = fut_hs - his_hs
```

Create a selection criteria xr.DataArray:

```
>>> from xarray import concat
>>> crit = concat((dtg, dhs), dim="criteria")
```

Finally, create clusters and select realization ids of reduced ensemble:

```
>>> ids, cluster, fig_data = kmeans_reduce_ensemble(
...     data=crit, method={"rsq_cutoff": 0.9}, random_state=42, make_graph=False
... )
>>> ids, cluster, fig_data = kmeans_reduce_ensemble(
...     data=crit, method={"rsq_optimize": None}, random_state=42, make_graph=True
... )
```

`xclim.ensembles.plot_rsqprofile(`*fig_data*`)`

Create an R^2 profile plot using kmeans_reduce_ensemble output.

The R^2 plot allows evaluation of the proportion of total uncertainty in the original ensemble that is provided by the reduced selected.

Examples

```
>>> from xclim.ensembles import kmeans_reduce_ensemble, plot_rsqprofile
>>> is_matplotlib_installed()
>>> crit = xr.open_dataset(path_to_ensemble_file).data
>>> ids, cluster, fig_data = kmeans_reduce_ensemble(
...     data=crit, method={"rsq_cutoff": 0.9}, random_state=42, make_graph=True
... )
>>> plot_rsqprofile(fig_data)
```

14.3.2 Ensemble Robustness metrics.

Robustness metrics are used to estimate the confidence of the climate change signal of an ensemble. This submodule is inspired by and tries to follow the guidelines of the IPCC, more specifically the 12th chapter of the Working Group 1's contribution to the AR5 [AR5WG1C12] (see box 12.1).

References

`xclim.ensembles.change_significance`(*fut: xr.DataArray | xr.Dataset, ref: xr.DataArray | xr.Dataset = None, test: str = 'ttest', **kwargs*) → tuple[xr.DataArray | xr.Dataset, xr.DataArray | xr.Dataset]

Robustness statistics qualifying how the members of an ensemble agree on the existence of change and on its sign.

Parameters

- **fut** (*Union[xr.DataArray, xr.Dataset]*) – Future period values along 'realization' and 'time' (..., nr, nt1) or if *ref* is None, Delta values along *realization* (..., nr).
- **ref** (*Union[xr.DataArray, xr.Dataset], optional*) – Reference period values along realization' and 'time' (..., nt2, nr). The size of the 'time' axis does not need to match the one of *fut*. But their 'realization' axes must be identical. If *None* (default), values of *fut* are assumed to be deltas instead of a distribution across the future period. *fut* and *ref* must be of the same type (Dataset or DataArray). If they are Dataset, they must have the same variables (name and coords).
- **test** (*{'ttest', 'welch-ttest', 'threshold', None}*) – Name of the statistical test used to determine if there was significant change. See notes.
- **kwargs** – Other arguments specific to the statistical test.

  For 'ttest' and 'welch-ttest':

  p_change
  [float (default][0.05)] p-value threshold for rejecting the hypothesis of no significant change.

  For 'threshold': (Only one of those must be given.)

  abs_thresh
  [float (no default)] Threshold for the (absolute) change to be considered significative.

  rel_thresh
  [float (no default, in [0, 1])] Threshold for the relative change (in reference to ref) to be significative. Only valid if *ref* is given.

Returns

- *change_frac* – The fraction of members that show significant change [0, 1]. Passing *test=None* yields change_frac = 1 everywhere. Same type as *fut*.
- *pos_frac* – The fraction of members showing significant change that show a positive change]0, 1]. Null values are returned where no members show significant change.

  The table below shows the coefficient needed to retrieve the number of members that have the indicated characteristics, by multiplying it to the total number of members (*fut.realization.size*).

	Significant change	Non-significant change
Any direction	change_frac	1 - change_frac
Positive change	pos_frac * change_frac	N.A.
Negative change	(1 - pos_frac) * change_frac	

Notes

Available statistical tests are :

'ttest' :
Single sample T-test. Same test as used by [tebaldi2011]. The future values are compared against the reference mean (over 'time'). Change is qualified as 'significant' when the test's p-value is below the user-provided *p_change* value.

'welch-ttest' :
Two-sided T-test, without assuming equal population variance. Same significance criterion as 'ttest'.

'threshold' :
Change is considered significative if the absolute delta exceeds a given threshold (absolute or relative).

None :
Significant change is not tested and, thus, members showing no change are included in the *sign_frac* output.

References

Example

This example computes the mean temperature in an ensemble and compares two time periods, qualifying significant change through a single sample T-test.

```
>>> from xclim import ensembles
>>> ens = ensembles.create_ensemble(temperature_datasets)
>>> tgmean = xclim.atmos.tg_mean(tas=ens.tas, freq="YS")
>>> fut = tgmean.sel(time=slice("2020", "2050"))
>>> ref = tgmean.sel(time=slice("1990", "2020"))
>>> chng_f, pos_f = ensembles.change_significance(fut, ref, test="ttest")
```

If the deltas were already computed beforehand, the 'threshold' test can still be used, here with a 2 K threshold.

```
>>> delta = fut.mean("time") - ref.mean("time")
>>> chng_f, pos_f = ensembles.change_significance(
...     delta, test="threshold", abs_thresh=2
... )
```

`xclim.ensembles.robustness_coefficient`(*fut: xr.DataArray | xr.Dataset*, *ref: xr.DataArray | xr.Dataset*) → xr.DataArray | xr.Dataset

Robustness coefficient quantifying the robustness of a climate change signal in an ensemble.

Taken from Knutti and Sedlacek (2013).

The robustness metric is defined as R = 1 A1 / A2 , where A1 is defined as the integral of the squared area between two cumulative density functions characterizing the individual model projections and the multi-model mean projection and A2 is the integral of the squared area between two cumulative density functions characterizing the multi-model mean projection and the historical climate. (Description taken from [knutti2013])

A value of R equal to one implies perfect model agreement. Higher model spread or smaller signal decreases the value of R.

Parameters

- **fut** (*Union[xr.DataArray, xr.Dataset]*) – Future ensemble values along 'realization' and 'time' (nr, nt). Can be a dataset, in which case the coeffcient is computed on each variables.
- **ref** (*Union[xr.DataArray, xr.Dataset]*) – Reference period values along 'time' (nt). Same type as *fut*.

Returns

R – The robustness coeffcient,]-inf, 1], float. Same type as *fut* or *ref*.

References

14.4 Indicator Tools

14.4.1 Indicators utilities

The *Indicator* class wraps indices computations with pre- and post-processing functionality. Prior to computations, the class runs data and metadata health checks. After computations, the class masks values that should be considered missing and adds metadata attributes to the object.

There are many ways to construct indicators. A good place to start is *this notebook*.

Dictionary and YAML parser

To construct indicators dynamically, xclim can also use dictionaries and parse them from YAML files. This is especially useful for generating whole indicator "submodules" from files. This functionality is inspired by the work of clix-meta.

YAML file structure

Indicator-defining yaml files are structured in the following way. Most entries of the *indicators* section are mirroring attributes of the `Indicator`, please refer to its documentation for more details on each.

```
module: <module name>  # Defaults to the file name
realm: <realm>  # If given here, applies to all indicators that do not already provide␣
↪it.
keywords: <keywords> # Merged with indicator-specific keywords (joined with a space)
references: <references> # Merged with indicator-specific references (joined with a new␣
↪line)
base: <base indicator class>  # Defaults to "Daily" and applies to all indicators that␣
↪do not give it.
doc: <module docstring>  # Defaults to a minimal header, only valid if the module doesn
↪'t already exists.
indicators:
  <identifier>:
    # From which Indicator to inherit
    base: <base indicator class>  # Defaults to module-wide base class
                                  # If the name startswith a '.', the base class is␣
↪taken from the current module (thus an indicator declared _above_)
                                  # Available classes are listed in `xclim.core.
↪indicator.registry` and `xclim.core.indicator.base_registry`.

    # General metadata, usually parsed from the `compute`'s docstring when possible.
    realm: <realm>  # defaults to module-wide realm. One of "atmos", "land", "seaIce",
↪"ocean".
    title: <title>
    abstract: <abstract>
    keywords: <keywords>  # Space-separated, merged to module-wide keywords.
    references: <references>  # newline-seperated, merged to module-wide references.
    notes: <notes>

    # Other options
    missing: <missing method name>
    missing_options:
        # missing options mapping
    allowed_periods: [<list>, <of>, <allowed>, <periods>]

    # Compute function
    compute: <function name>  # Referring to a function in the passed indices module,␣
↪xclim.indices.generic or xclim.indices
    input:  # When "compute" is a generic function this is a mapping from argument
            # name to what CMIP6/xclim variable is expected. This will allow for
            # declaring expected input units and have a CF metadata check on the inputs.
            # Can also be used to modify the expected variable, as long as it has
            # the same units. Ex: tas instead of tasmin.
      <var name in compute> : <variable official name>
      ...
    parameters:
     <param name>: <param data>  # Simplest case, to inject parameters in the compute␣
↪function.
     <param name>:  # To change parameters metadata or to declare units when "compute"␣
↪is a generic function.
```

(continues on next page)

(continued from previous page)

```
        units: <param units>  # Only valid if "compute" points to a generic function
        default : <param default>
        description: <param description>
    ...
  ...  # and so on.
```

All fields are optional. Other fields found in the yaml file will trigger errors in xclim. In the following, the section under *<identifier>* is refered to as *data*. When creating indicators from a dictionary, with `Indicator.from_dict()`, the input dict must follow the same structure of *data*.

The resulting yaml file can be validated using the provided schema (in xclim/data/schema.yml) and the yamale tool. See the "Extending xclim" notebook for more info.

Inputs

As xclim has strict definitions of possible input variables (see `xclim.core.utils.variables`), the mapping of *data.input* simply links an argument name from the function given in "compute" to one of those official variables.

class `xclim.core.indicator.Parameter`(*kind: ~xclim.core.utils.InputKind, default: ~typing.Any, description: str = '', units: str = <class 'xclim.core.indicator._empty'>, choices: set = <class 'xclim.core.indicator._empty'>, value: ~typing.Any = <class 'xclim.core.indicator._empty'>*)

Bases: `object`

Class for storing an indicator's controllable parameter.

For retrocompatibility, this class implements a "getitem" and a special "contains".

Example

```
>>> p = Parameter(InputKind.NUMBER, default=2, description="A simple number")
>>> p.units is Parameter._empty  # has not been set
True
>>> "units" in p  # Easier/retrocompatible way to test if units are set
False
>>> p.description
'A simple number'
>>> p["description"]  # Same as above, for convenience.
'A simple number'
```

`default`

alias of `_empty`

`update`(*other: dict*) → None

Update a parameter's values from a dict.

classmethod `is_parameter_dict`(*other: dict*) → bool

Return whether indicator has a parameter dictionary.

`asdict()` → dict

Format indicators as a dictionary.

`property injected: bool`

Indicate whether values are injected.

`class xclim.core.indicator.IndicatorRegistrar`

Bases: `object`

Climate Indicator registering object.

`classmethod get_instance()`

Return first found instance.

Raises *ValueError* if no instance exists.

`class xclim.core.indicator.Indicator(`***kwds*`)`

Bases: *`IndicatorRegistrar`*

Climate indicator base class.

Climate indicator object that, when called, computes an indicator and assigns its output a number of CF-compliant attributes. Some of these attributes can be *templated*, allowing metadata to reflect the value of call arguments.

Instantiating a new indicator returns an instance but also creates and registers a custom subclass in `xclim.core.indicator.registry`.

Attributes in *Indicator.cf_attrs* will be formatted and added to the output variable(s). This attribute is a list of dictionaries. For convenience and retro-compatibility, standard CF attributes (names listed in *`xclim.core.indicator.Indicator._cf_names`*) can be passed as strings or list of strings directly to the indicator constructor.

A lot of the Indicator's metadata is parsed from the underlying *compute* function's docstring and signature. Input variables and parameters are listed in *`xclim.core.indicator.Indicator.parameters`*, while parameters that will be injected in the compute function are in *`xclim.core.indicator.Indicator.injected_parameters`*. Both are simply views of *`xclim.core.indicator.Indicator._all_parameters`*.

Compared to their base *compute* function, indicators add the possibility of using dataset as input, with the injected argument *ds* in the call signature. All arguments that were indicated by the compute function to be variables (DataArrays) through annotations will be promoted to also accept strings that correspond to variable names in the *ds* dataset.

Parameters

- **identifier** (*str*) – Unique ID for class registry, should be a valid slug.
- **realm** (*{'atmos', 'seaIce', 'land', 'ocean'}*) – General domain of validity of the indicator. Indicators created outside xclim.indicators must set this attribute.
- **compute** (*func*) – The function computing the indicators. It should return one or more DataArray.
- **cf_attrs** (*list of dicts*) – Attributes to be formatted and added to the computation's output. See *`xclim.core.indicator.Indicator.cf_attrs`*.
- **title** (*str*) – A succinct description of what is in the computed outputs. Parsed from *compute* docstring if None (first paragraph).
- **abstract** (*str*) – A long description of what is in the computed outputs. Parsed from *compute* docstring if None (second paragraph).

- **keywords** (*str*) – Comma separated list of keywords. Parsed from *compute* docstring if None (from a "Keywords" section).
- **references** (*str*) – Published or web-based references that describe the data or methods used to produce it. Parsed from *compute* docstring if None (from the "References" section).
- **notes** (*str*) – Notes regarding computing function, for example the mathematical formulation. Parsed from *compute* docstring if None (form the "Notes" section).
- **src_freq** (*str, sequence of strings, optional*) – The expected frequency of the input data. Can be a list for multiple frequencies, or None if irrelevant.
- **context** (*str*) – The *pint* unit context, for example use 'hydro' to allow conversion from kg m-2 s-1 to mm/day.

Notes

All subclasses created are available in the *registry* attribute and can be used to define custom subclasses or parse all available instances.

`cf_attrs: Sequence[Mapping[str, Any]] = None`

A list of metadata information for each output of the indicator.

It minimally contains a "var_name" entry, and may contain : "standard_name", "long_name", "units", "cell_methods", "description" and "comment" on official xclim indicators. Other fields could also be present if the indicator was created from outside xclim.

var_name:
Output variable(s) name(s).

standard_name:
Variable name, must be in the CF standard names table (this is not checked).

long_name:
Descriptive variable name. Parsed from *compute* docstring if not given. (first line after the output dtype, only works on single output function).

units:
Representative units of the physical quantity.

cell_methods:
List of blank-separated words of the form "name: method". Must respect the CF-conventions and vocabulary (not checked).

description:
Sentence(s) meant to clarify the qualifiers of the fundamental quantities, such as which surface a quantity is defined on or what the flux sign conventions are.

comment:
Miscellaneous information about the data or methods used to produce it.

`classmethod from_dict(`*data: dict*, *identifier: str*, *module: str | None = None*`)`

Create an indicator subclass and instance from a dictionary of parameters.

Most parameters are passed directly as keyword arguments to the class constructor, except:

- "base" : A subclass of Indicator or a name of one listed in `xclim.core.indicator.registry` or `xclim.core.indicaotr.base_registry`. When passed, it acts as if *from_ dict* was called on that class instead.

- "compute" : A string function name translates to a `xclim.indices.generic` or `xclim.indices` function.

  Parameters

  - **data** (*dict*) – The exact structure of this dictionary is detailed in the submodule documentation.
  - **identifier** (*str*) – The name of the subclass and internal indicator name.
  - **module** (*str*) – The module name of the indicator. This is meant to be used only if the indicator is part of a dynamically generated submodule, to override the module of the base class.

`classmethod translate_attrs(`*locale: str | Sequence[str], fill_missing: bool = True*`)`

Return a dictionary of unformated translated translatable attributes.

Translatable attributes are defined in `xclim.core.locales.TRANSLATABLE_ATTRS`.

Parameters

- **locale** (*Union[str, Sequence[str]]*) – The POSIX name of the locale or a tuple of a locale name and a path to a json file defining the translations. See *xclim.locale* for details.
- **fill_missing** (*bool*) – If True (default) fill the missing attributes by their english values.

`json(`*args=None*`)`

Return a serializable dictionary representation of the class.

Parameters

args (*mapping, optional*) – Arguments as passed to the call method of the indicator. If not given, the default arguments will be used when formatting the attributes.

Notes

This is meant to be used by a third-party library wanting to wrap this class into another interface.

`static compute(`**args, **kwds*`)`

Compute the indicator.

This would typically be a function from *xclim.indices*.

`cfcheck(`***das*`)`

Compare metadata attributes to CF-Convention standards.

Default cfchecks use the specifications in *xclim.core.utils.VARIABLES*, assuming the indicator's inputs are using the CMIP6/xclim variable names correctly. Variables absent from these default specs are silently ignored.

When subclassing this method, use functions decorated using *xclim.core.options.cfcheck*.

`datacheck(`***das*`)`

Verify that input data is valid.

When subclassing this method, use functions decorated using *xclim.core.options.datacheck*.

For example, checks could include:

- assert no precipitation is negative

- assert no temperature has the same value 5 days in a row

This base datacheck checks that the input data has a valid sampling frequency, as given in self.src_freq.

`property n_outs`

Return the length of all cf_attrs.

`property parameters`

Create a dictionary of controllable parameters.

Similar to *Indicator._all_parameters*, but doesn't include injected parameters.

`property injected_parameters`

Return a dictionary of all injected parameters.

Opposite of *Indicator.parameters()*.

`class xclim.core.indicator.ResamplingIndicator(**kwds)`

Bases: *Indicator*

Indicator that performs a resampling computation.

Compared to the base Indicator, this adds the handling of missing data, and the check of allowed periods.

Parameters

- **missing** (*{any, wmo, pct, at_least_n, skip, from_context}*) – The name of the missing value method. See *xclim.core.missing.MissingBase* to create new custom methods. If None, this will be determined by the global configuration (see *xclim.set_options*). Defaults to "from_context".
- **missing_options** (*dict, None*) – Arguments to pass to the *missing* function. If None, this will be determined by the global configuration.
- **allowed_periods** (*Sequence[str], optional*) – A list of allowed periods, i.e. base parts of the *freq* parameter. For example, indicators meant to be computed annually only will have *allowed_periods=["A"]. None* means "any period" or that the indicator doesn't take a *freq* argument.

`class xclim.core.indicator.ResamplingIndicatorWithIndexing(**kwds)`

Bases: *ResamplingIndicator*

Resampling indicator that also injects "indexer" kwargs to subset the inputs before computation.

`class xclim.core.indicator.Daily(**kwds)`

Bases: *ResamplingIndicator*

Class for daily inputs and resampling computes.

`class xclim.core.indicator.Hourly(**kwds)`

Bases: *ResamplingIndicator*

Class for hourly inputs and resampling computes.

`xclim.core.indicator.add_iter_indicators(module)`

Create an iterable of loaded indicators.

`xclim.core.indicator.build_indicator_module(name: str, objs: Mapping[str, Indicator], doc: str | None = None) → ModuleType`

Create or update a module from imported objects.

The module is inserted as a submodule of `xclim.indicators`.

Parameters

- **name** (*str*) – New module name. If it already exists, the module is extended with the passed objects, overwriting those with same names.
- **objs** (*dict*) – Mapping of the indicators to put in the new module. Keyed by the name they will take in that module.
- **doc** (*str*) – Docstring of the new module. Defaults to a simple header. Invalid if the module already exists.

Returns

ModuleType – A indicator module built from a mapping of Indicators.

`xclim.core.indicator.build_indicator_module_from_yaml`(*filename: PathLike*, *name: str | None = None*, *indices: Mapping[str, Callable] | ModuleType | PathLike | None = None*, *translations: dict[str, dict | PathLike] | None = None*, *mode: str = 'raise'*, *encoding: str = 'UTF8'*) → ModuleType

Build or extend an indicator module from a YAML file.

The module is inserted as a submodule of `xclim.indicators`. When given only a base filename (no 'yml' extesion), this tries to find custom indices in a module of the same name (*.py) and translations in json files (*.<lang>.json), see Notes.

Parameters

- **filename** (*PathLike*) – Path to a YAML file or to the stem of all module files. See Notes for behaviour when passing a basename only.
- **name** (*str, optional*) – The name of the new or existing module, defaults to the basename of the file. (e.g: *atmos.yml* -> *atmos*)
- **indices** (*Mapping of callables or module or path, optional*) – A mapping or module of indice functions or a python file declaring such a file. When creating the indicator, the name in the *index_function* field is first sought here, then the indicator class will search in xclim.indices.generic and finally in xclim.indices.
- **translations** (*Mapping of dicts or path, optional*) – Translated metadata for the new indicators. Keys of the mapping must be 2-char language tags. Values can be translations dictionaries as defined in *Internationalization*. They can also be a path to a json file defining the translations.
- **mode** (*{'raise', 'warn', 'ignore'}*) – How to deal with broken indice definitions.
- **encoding** (*str*) – The encoding used to open the *.yaml* and *.json* files. It defaults to UTF-8, overriding python's mechanism which is machine dependent.

Returns

ModuleType – A submodule of *pym:mod:`xclim.indicators.*

Notes

When the given *filename* has no suffix (usually '.yaml' or '.yml'), the function will try to load custom indice definitions from a file with the same name but with a *.py* extension. Similarly, it will try to load translations in *.<lang>.json files, where *<lang>* is the IETF language tag.

For example. a set of custom indicators could be fully described by the following files:

- *example.yml* : defining the indicator's metadata.
- *example.py* : defining a few indice functions.
- *example.fr.json* : French translations
- *example.tlh.json* : Klingon translations.

See also:

`The`

14.5 Unit Handling module

14.5.1 Units handling submodule

Pint is used to define the *units UnitRegistry* and *xclim.units.core* defines most unit handling methods.

`xclim.core.units.check_units(`*val: str | int | float | None*, *dim: str | None*`)` → None

Check units for appropriate convention compliance.

`xclim.core.units.convert_units_to(`*source: str | xr.DataArray | Any*, *target: str | xr.DataArray | Any*, *context: str | None = None*`)` → xr.DataArray | float | int | str | Any

Convert a mathematical expression into a value with the same units as a DataArray.

Parameters

- **source** (*Union[str, xr.DataArray, Any]*) – The value to be converted, e.g. '4C' or '1 mm/d'.
- **target** (*Union[str, xr.DataArray, Any]*) – Target array of values to which units must conform.
- **context** (*str, optional*) – The unit definition context. Default: None.

Returns

Union[xr.DataArray, float, int, str, Any] – The source value converted to target's units.

`xclim.core.units.declare_units(`***units_by_name*`)` → Callable

Create a decorator to check units of function arguments.

The decorator checks that input and output values have units that are compatible with expected dimensions. It also stores the input units as a 'in_units' attribute.

Parameters

units_by_name (*Mapping[str, str]*) – Mapping from the input parameter names to their units or dimensionality ("[...]").

Examples

In the following function definition:

```
@declare_units(tas=["temperature"])
def func(tas):
    ...
```

The decorator will check that *tas* has units of temperature (C, K, F).

`xclim.core.units.infer_sampling_units(`*da: xr.DataArray*, *deffreq: str | None = 'D'*, *dim: str = 'time'*`)` → tuple[int, str]

Infer a multiplicator and the units corresponding to one sampling period.

Parameters

- **da** (*xr.DataArray*) – A DataArray from which to take coordinate *dim*.
- **deffreq** (*str*) – If no frequency is inferred from *da[dim]*, take this one.
- **dim** (*str*) – Dimension from which to infer the frequency.

Raises

`ValueError` – If the frequency has no exact corresponding units.

Returns

- **m** (*int*) – The magnitude (number of base periods per period)
- **u** (*str*) – Units as a string, understandable by pint.

`xclim.core.units.pint2cfunits(`*value: UnitDefinition*`)` → str

Return a CF-compliant unit string from a *pint* unit.

Parameters

value (*pint.Unit*) – Input unit.

Returns

out (*str*) – Units following CF-Convention, using symbols.

`xclim.core.units.pint_multiply(`*da: xr.DataArray*, *q: Any*, *out_units: str | None = None*`)`

Multiply xarray.DataArray by pint.Quantity.

Parameters

- **da** (*xr.DataArray*) – Input array.
- **q** (*pint.Quantity*) – Multiplicative factor.
- **out_units** (*Optional[str]*) – Units the output array should be converted into.

`xclim.core.units.rate2amount(`*rate: DataArray*, *dim: str = 'time'*, *out_units: Optional[str] = None*`)` → DataArray

Convert a rate variable to an amount by multiplying by the sampling period length.

If the sampling period length cannot be inferred, the rate values are multiplied by the duration between their time coordinate and the next one. The last period is estimated with the duration of the one just before.

This is the inverse operation of `amount2rate()`.

Parameters

- **rate** (*xr.DataArray*) – "Rate" variable, with units of "amount" per time. Ex: Precipitation in "mm / d".
- **dim** (*str*) – The time dimension.
- **out_units** (*str, optional*) – Output units to convert to.

Returns
xr.DataArray

Examples

The following converts a daily array of precipitation in mm/h to the daily amounts in mm.

```
>>> time = xr.cftime_range("2001-01-01", freq="D", periods=365)
>>> pr = xr.DataArray(
...     [1] * 365, dims=("time",), coords={"time": time}, attrs={"units": "mm/h"}
... )
>>> pram = rate2amount(pr)
>>> pram.units
'mm'
>>> float(pram[0])
24.0
```

Also works if the time axis is irregular : the rates are assumed constant for the whole period starting on the values timestamp to the next timestamp.

```
>>> time = time[[0, 9, 30]]  # The time axis is Jan 1st, Jan 10th, Jan 31st
>>> pr = xr.DataArray(
...     [1] * 3, dims=("time",), coords={"time": time}, attrs={"units": "mm/h"}
... )
>>> pram = rate2amount(pr)
>>> pram.values
array([216., 504., 504.])
```

Finally, we can force output units:

```
>>> pram = rate2amount(pr, out_units="pc")  # Get rain amount in parsecs. Why not.
>>> pram.values
array([7.00008327e-18, 1.63335276e-17, 1.63335276e-17])
```

`xclim.core.units.str2pint`(*val: str*)

Convert a string to a pint.Quantity, splitting the magnitude and the units.

Parameters
val (*str*) – A quantity in the form "[{magnitude}]{units}", where magnitude is castable to a float and units is understood by *units2pint*.

Returns
pint.Quantity – Magnitude is 1 if no magnitude was present in the string.

`xclim.core.units.to_agg_units`(*out: DataArray, orig: DataArray, op: str, dim: str = 'time'*) → DataArray

Set and convert units of an array after an aggregation operation along the sampling dimension (time).

Parameters

- **out** (*xr.DataArray*) – The output array of the aggregation operation, no units operation done yet.
- **orig** (*xr.DataArray*) – The original array before the aggregation operation, used to infer the sampling units and get the variable units.
- **op** (*{'count', 'prod', 'delta_prod'}*) – The type of aggregation operation performed. The special "delta_*" ops are used with temperature units needing conversion to their "delta" counterparts (e.g. degree days)
- **dim** (*str*) – The time dimension along which the aggregation was performed.

Examples

Take a daily array of temperature and count number of days above a threshold. *to_agg_units* will infer the units from the sampling rate along "time", so we ensure the final units are correct.

```
>>> time = xr.cftime_range("2001-01-01", freq="D", periods=365)
>>> tas = xr.DataArray(
...     np.arange(365),
...     dims=("time",),
...     coords={"time": time},
...     attrs={"units": "degC"},
... )
>>> cond = tas > 100  # Which days are boiling
>>> Ndays = cond.sum("time")  # Number of boiling days
>>> Ndays.attrs.get("units")
None
>>> Ndays = to_agg_units(Ndays, tas, op="count")
>>> Ndays.units
'd'
```

Similarly, here we compute the total heating degree-days but we have weekly data: >>> time = xr.cftime_range("2001-01-01", freq="7D", periods=52) >>> tas = xr.DataArray(… np.arange(52) + 10, … dims=("time",), … coords={"time": time}, … attrs={"units": "degC"}, …) >>> degdays = (… (tas - 16).clip(0).sum("time") …) # Integral of temperature above a threshold >>> degdays = to_agg_units(degdays, tas, op="delta_prod") >>> degdays.units 'week delta_degC'

Which we can always convert to the more common "K days":

```
>>> degdays = convert_units_to(degdays, "K days")
>>> degdays.units
'K d'
```

`xclim.core.units.units2pint`(*value: xr.DataArray | str | units.Quantity*) → Unit

Return the pint Unit for the DataArray units.

Parameters

value (*Union[xr.DataArray, str. pint.Quantity]*) – Input data array or string representing a unit (with no magnitude).

Returns

pint.unit.UnitDefinition – Units of the data array.

14.6 Other Utilities

14.6.1 Calendar handling utilities

Helper function to handle dates, times and different calendars with xarray.

`xclim.core.calendar.DayOfYearStr`

Type annotation for strings representing dates without a year (MM-DD).

alias of `str`

`xclim.core.calendar.adjust_doy_calendar`(*source: xr.DataArray*, *target: xr.DataArray | xr.Dataset*) → xr.DataArray

Interpolate from one set of dayofyear range to another calendar.

Interpolate an array defined over a *dayofyear* range (say 1 to 360) to another *dayofyear* range (say 1 to 365).

Parameters

- **source** (*xr.DataArray*) – Array with *dayofyear* coordinate.
- **target** (*xr.DataArray or xr.Dataset*) – Array with *time* coordinate.

Returns

xr.DataArray – Interpolated source array over coordinates spanning the target *dayofyear* range.

`xclim.core.calendar.cfindex_end_time`(*cfindex: CFTimeIndex*, *freq: str*) → CFTimeIndex

Get the end of a period for a pseudo-period index.

As we are using datetime indices to stand in for period indices, assumptions regarding the period are made based on the given freq. IMPORTANT NOTE: this function cannot be used on greater-than-day freq that start at the beginning of a month, e.g. 'MS', 'QS', 'AS' – this mirrors pandas behavior.

Parameters

- **cfindex** (*CFTimeIndex*) – CFTimeIndex as a proxy representation for CFPeriodIndex
- **freq** (*str*) – String specifying the frequency/offset such as 'MS', '2D', 'H', or '3T'

Returns

CFTimeIndex – The ending datetimes of periods inferred from dates and freq

`xclim.core.calendar.cfindex_start_time`(*cfindex: CFTimeIndex*, *freq: str*) → CFTimeIndex

Get the start of a period for a pseudo-period index.

As we are using datetime indices to stand in for period indices, assumptions regarding the period are made based on the given freq. IMPORTANT NOTE: this function cannot be used on greater-than-day freq that start at the beginning of a month, e.g. 'MS', 'QS', 'AS' – this mirrors pandas behavior.

Parameters

- **cfindex** (*CFTimeIndex*) – CFTimeIndex as a proxy representation for CFPeriodIndex
- **freq** (*str*) – String specifying the frequency/offset such as 'MS', '2D', 'H', or '3T'

Returns

CFTimeIndex – The starting datetimes of periods inferred from dates and freq `xclim.core.calendar.cftime_end_time(`*date: datetime*, *freq: str*`)` → datetime

Get the cftime.datetime for the end of a period.

As we are not supplying actual period objects, assumptions regarding the period are made based on the given freq. IMPORTANT NOTE: this function cannot be used on greater-than-day freq that start at the beginning of a month, e.g. 'MS', 'QS', 'AS' – this mirrors pandas behavior.

Parameters

- **date** (*cftime.datetime*) – The original datetime object as a proxy representation for period.
- **freq** (*str*) – String specifying the frequency/offset such as 'MS', '2D', 'H', or '3T'

Returns

cftime.datetime – The ending datetime of the period inferred from date and freq.

`xclim.core.calendar.cftime_start_time(`*date: datetime*, *freq: str*`)` → datetime

Get the cftime.datetime for the start of a period.

As we are not supplying actual period objects, assumptions regarding the period are made based on the given freq. IMPORTANT NOTE: this function cannot be used on greater-than-day freq that start at the beginning of a month, e.g. 'MS', 'QS', 'AS' – this mirrors pandas behavior.

Parameters

- **date** (*cftime.datetime*) – The original datetime object as a proxy representation for period.
- **freq** (*str*) – String specifying the frequency/offset such as 'MS', '2D', 'H', or '3T'

Returns

cftime.datetime – The starting datetime of the period inferred from date and freq.

`xclim.core.calendar.climatological_mean_doy(`*arr: xr.DataArray*, *window: int = 5*`)` → tuple[xr.DataArray, xr.DataArray]

Calculate the climatological mean and standard deviation for each day of the year.

Parameters

- **arr** (*xarray.DataArray*) – Input array.
- **window** (*int*) – Window size in days.

Returns

xarray.DataArray, xarray.DataArray – Mean and standard deviation.

`xclim.core.calendar.compare_offsets(`*freqA: str*, *op: str*, *freqB: str*`)` → bool

Compare offsets string based on their approximate length, according to a given operator.

Offset are compared based on their length approximated for a period starting after 1970-01-01 00:00:00. If the offsets are from the same category (same first letter), only the multiplicator prefix is compared (QS-DEC == QS-JAN, MS < 2MS). "Business" offsets are not implemented.

Parameters

- **freqA** (*str*) – RHS Date offset string ('YS', '1D', 'QS-DEC', …)
- **op** (*{'<', '<=', '==', '>', '>=', '!='}*) – Operator to use.
- **freqB** (*str*) – LHS Date offset string ('YS', '1D', 'QS-DEC', …)

Returns

bool – freqA op freqB

xclim.core.calendar.convert_calendar(*source: xr.DataArray | xr.Dataset*, *target: xr.DataArray | str*, *align_on: str | None = None*, *missing: Any | None = None*, *dim: str = 'time'*) → xr.DataArray | xr.Dataset

Convert a DataArray/Dataset to another calendar using the specified method.

Only converts the individual timestamps, does not modify any data except in dropping invalid/surplus dates or inserting missing dates.

If the source and target calendars are either no_leap, all_leap or a standard type, only the type of the time array is modified. When converting to a leap year from a non-leap year, the 29th of February is removed from the array. In the other direction and if *target* is a string, the 29th of February will be missing in the output, unless *missing* is specified, in which case that value is inserted.

For conversions involving *360_day* calendars, see Notes.

This method is safe to use with sub-daily data as it doesn't touch the time part of the timestamps.

Parameters

- **source** (*xr.DataArray*) – Input array/dataset with a time coordinate of a valid dtype (datetime64 or a cftime.datetime).
- **target** (*Union[xr.DataArray, str]*) – Either a calendar name or the 1D time coordinate to convert to. If an array is provided, the output will be reindexed using it and in that case, days in *target* that are missing in the converted *source* are filled by *missing* (which defaults to NaN).
- **align_on** (*{None, 'date', 'year', 'random'}*) – Must be specified when either source or target is a *360_day* calendar, ignored otherwise. See Notes.
- **missing** (*Optional[any]*) – A value to use for filling in dates in the target that were missing in the source. If *target* is a string, default (None) is not to fill values. If it is an array, default is to fill with NaN.
- **dim** (*str*) – Name of the time coordinate.

Returns

Union[xr.DataArray, xr.Dataset] – Copy of source with the time coordinate converted to the target calendar. If *target* is given as an array, the output is reindexed to it, with fill value *missing*. If *target* was a string and *missing* was None (default), invalid dates in the new calendar are dropped, but missing dates are not inserted. If *target* was a string and *missing* was given, then start, end and frequency of the new time axis are inferred and the output is reindexed to that a new array.

Notes

If one of the source or target calendars is *360_day*, *align_on* must be specified and two options are offered.

"year"

The dates are translated according to their rank in the year (dayofyear), ignoring their original month and day information, meaning that the missing/surplus days are added/removed at regular intervals.

From a *360_day* to a standard calendar, the output will be missing the following dates (day of year in parenthesis):

To a leap year:

January 31st (31), March 31st (91), June 1st (153), July 31st (213), September 31st (275) and November 30th (335).

To a non-leap year:
February 6th (36), April 19th (109), July 2nd (183), September 12th (255), November 25th (329).

From standard calendar to a '360_day', the following dates in the source array will be dropped:

From a leap year:
January 31st (31), April 1st (92), June 1st (153), August 1st (214), September 31st (275), December 1st (336)

From a non-leap year:
February 6th (37), April 20th (110), July 2nd (183), September 13th (256), November 25th (329)

This option is best used on daily and subdaily data.

"date"
The month/day information is conserved and invalid dates are dropped from the output. This means that when converting from a *360_day* to a standard calendar, all 31st (Jan, March, May, July, August, October and December) will be missing as there is no equivalent dates in the *360_day* and the 29th (on non-leap years) and 30th of February will be dropped as there are no equivalent dates in a standard calendar.

This option is best used with data on a frequency coarser than daily.

"random"
Similar to "year", each day of year of the source is mapped to another day of year of the target. However, instead of having always the same missing days according the source and target years, here 5 days are chosen randomly, one for each fifth of the year. However, February 29th is always missing when converting to a leap year, or its value is dropped when converting from a leap year. This is similar to method used in the [LOCA] dataset.

This option best used on daily data.

References

Examples

This method does not try to fill the missing dates other than with a constant value, passed with *missing*. In order to fill the missing dates with interpolation, one can simply use xarray's method:

```
>>> tas_nl = convert_calendar(tas, "noleap")  # For the example
>>> with_missing = convert_calendar(tas_nl, "standard", missing=np.NaN)
>>> out = with_missing.interpolate_na("time", method="linear")
```

Here, if Nans existed in the source data, they will be interpolated too. If that is, for some reason, not wanted, the workaround is to do:

```
>>> mask = convert_calendar(tas_nl, "standard").notnull()
>>> out2 = out.where(mask)
```

`xclim.core.calendar.date_range(`**args*, *calendar: str = 'default'*, ***kwargs*`)` → pd.DatetimeIndex | CFTimeIndex

Wrap pd.date_range (if calendar == 'default') or xr.cftime_range (otherwise).

`xclim.core.calendar.date_range_like`(*source: DataArray*, *calendar: str*) → DataArray

Generate a datetime array with the same frequency, start and end as another one, but in a different calendar.

Parameters

- **source** (*xr.DataArray*) – 1D datetime coordinate DataArray
- **calendar** (*str*) – New calendar name.

Raises

`ValueError` – If the source's frequency was not found.

Returns

xr.DataArray –

1D datetime coordinate with the same start, end and frequency as the source, but in the new calendar.
The start date is assumed to exist in the target calendar. If the end date doesn't exist, the code tries 1 and 2 calendar days before. Exception when the source is in 360_day and the end of the range is the 30th of a 31-days month, then the 31st is appended to the range.

`xclim.core.calendar.datetime_to_decimal_year`(*times: DataArray*, *calendar: str = ''*) → DataArray

Convert a datetime xr.DataArray to decimal years according to its calendar or the given one.

Decimal years are the number of years since 0001-01-01 00:00:00 AD. Ex: '2000-03-01 12:00' is 2000.1653 in a standard calendar, 2000.16301 in a "noleap" or 2000.16806 in a "360_day".

`xclim.core.calendar.days_in_year`(*year: int*, *calendar: str = 'default'*) → int

Return the number of days in the input year according to the input calendar.

`xclim.core.calendar.days_since_to_doy`(*da: xr.DataArray*, *start: DayOfYearStr | None = None*, *calendar: str | None = None*) → xr.DataArray

Reverse the conversion made by `doy_to_days_since()`.

Converts data given in days since a specific date to day-of-year.

Parameters

- **da** (*xr.DataArray*) – The result of `doy_to_days_since()`.
- **start** (*DateOfYearStr, optional*) – *da* is considered as days since that start date (in the year of the time index). If None (default), it is read from the attributes.
- **calendar** (*str, optional*) – Calendar the "days since" were computed in. If None (default), it is read from the attributes.

Returns

xr.DataArray – Same shape as *da*, values as *day of year*.

Examples

```
>>> from xarray import DataArray
>>> time = date_range("2020-07-01", "2021-07-01", freq="AS-JUL")
>>> da = DataArray(
...     [-86, 92],
...     dims=("time",),
...     coords={"time": time},
...     attrs={"units": "days since 10-02"},
... )
>>> days_since_to_doy(da).values
array([190, 2])
```

`xclim.core.calendar.doy_to_days_since`(*da: xr.DataArray*, *start: DayOfYearStr | None = None*, *calendar: str | None = None*) → xr.DataArray

Convert day-of-year data to days since a given date.

This is useful for computing meaningful statistics on doy data.

Parameters

- **da** (*xr.DataArray*) – Array of "day-of-year", usually int dtype, must have a *time* dimension. Sampling frequency should be finer or similar to yearly and coarser then daily.
- **start** (*date of year str, optional*) – A date in "MM-DD" format, the base day of the new array. If None (default), the *time* axis is used. Passing *start* only makes sense if *da* has a yearly sampling frequency.
- **calendar** (*str, optional*) – The calendar to use when computing the new interval. If None (default), the calendar attribute of the data or of its *time* axis is used. All time coordinates of *da* must exist in this calendar. No check is done to ensure doy values exist in this calendar.

Returns

xr.DataArray – Same shape as *da*, int dtype, day-of-year data translated to a number of days since a given date. If start is not None, there might be negative values.

Notes

The time coordinates of *da* are considered as the START of the period. For example, a doy value of 350 with a timestamp of '2020-12-31' is understood as '2021-12-16' (the 350th day of 2021). Passing *start=None*, will use the time coordinate as the base, so in this case the converted value will be 350 "days since time coordinate".

Examples

```
>>> from xarray import DataArray
>>> time = date_range("2020-07-01", "2021-07-01", freq="AS-JUL")
>>> # July 8th 2020 and Jan 2nd 2022
>>> da = DataArray([190, 2], dims=("time",), coords={"time": time})
>>> # Convert to days since Oct. 2nd, of the data's year.
>>> doy_to_days_since(da, start="10-02").values
array([-86, 92])
```

`xclim.core.calendar.ensure_cftime_array(`*time: Sequence*`)` → ndarray

Convert an input 1D array to a numpy array of cftime objects.

Python's datetime are converted to cftime.DatetimeGregorian ("standard" calendar).

Raises ValueError when unable to cast the input.

`xclim.core.calendar.get_calendar(`*obj: Any*, *dim: str = 'time'*`)` → str

Return the calendar of an object.

Parameters

- **obj** (*Any*) – An object defining some date. If *obj* is an array/dataset with a datetime coordinate, use *dim* to specify its name. Values must have either a datetime64 dtype or a cftime dtype. *obj* can also be a python datetime.datetime, a cftime object or a pandas Timestamp or an iterable of those, in which case the calendar is inferred from the first value.
- **dim** (*str*) – Name of the coordinate to check (if *obj* is a DataArray or Dataset).

Raises

`ValueError` – If no calendar could be inferred.

Returns

str – The cftime calendar name or "default" when the data is using numpy's or python's datetime types. Will always return "standard" instead of "gregorian", following CF conventions 1.9.

`xclim.core.calendar.interp_calendar(`*source: xr.DataArray | xr.Dataset*, *target: xr.DataArray*, *dim: str = 'time'*`)` → xr.DataArray | xr.Dataset

Interpolates a DataArray/Dataset to another calendar based on decimal year measure.

Each timestamp in source and target are first converted to their decimal year equivalent then source is interpolated on the target coordinate. The decimal year is the number of years since 0001-01-01 AD. Ex: '2000-03-01 12:00' is 2000.1653 in a standard calendar or 2000.16301 in a 'noleap' calendar.

This method should be used with daily data or coarser. Sub-daily result will have a modified day cycle.

Parameters

- **source** (*Union[xr.DataArray, xr.Dataset]*) – The source data to interpolate, must have a time coordinate of a valid dtype (np.datetime64 or cftime objects)
- **target** (*xr.DataArray*) – The target time coordinate of a valid dtype (np.datetime64 or cftime objects)
- **dim** (*str*) – The time coordinate name.

Returns

Union[xr.DataArray, xr.Dataset] – The source interpolated on the decimal years of target,

`xclim.core.calendar.parse_offset(`*freq: str*`)` → Sequence[str]

Parse an offset string.

Parse a frequency offset and, if needed, convert to cftime-compatible components.

Parameters

freq (*str*) – Frequency offset.

Returns

multiplicator (int), offset base (str), is start anchored (bool), anchor (str or None) –

"[n]W" is always replaced with "[7n]D", as xarray doesn't support "W" for cftime indexes. "Y" is always replaced with "A".

`xclim.core.calendar.percentile_doy(`*arr: xr.DataArray, window: int = 5, per: float | Sequence[float] = 10.0, alpha: float = 0.3333333333333333, beta: float = 0.3333333333333333, copy: bool = True*`)` → *PercentileDataArray*

Percentile value for each day of the year.

Return the climatological percentile over a moving window around each day of the year. Different quantile estimators can be used by specifying *alpha* and *beta* according to specifications given by [HyndmanFan]. The default definition corresponds to method 8, which meets multiple desirable statistical properties for sample quantiles. Note that *numpy.percentile* corresponds to method 7, with alpha and beta set to 1.

Parameters

- **arr** (*xr.DataArray*) – Input data, a daily frequency (or coarser) is required.
- **window** (*int*) – Number of time-steps around each day of the year to include in the calculation.
- **per** (*float or sequence of floats*) – Percentile(s) between [0, 100]
- **alpha** (*float*) – Plotting position parameter.
- **beta** (*float*) – Plotting position parameter.
- **copy** (*bool*) – If True (default) the input array will be deep copied. It's a necessary step to keep the data integrity but it can be costly. If False, no copy is made of the input array. It will be mutated and rendered unusable but performances may significantly improve. Put this flag to False only if you understand the consequences.

Returns

xr.DataArray – The percentiles indexed by the day of the year. For calendars with 366 days, percentiles of doys 1-365 are interpolated to the 1-366 range.

References

`xclim.core.calendar.resample_doy(`*doy: xr.DataArray, arr: xr.DataArray | xr.Dataset*`)` → xr.DataArray

Create a temporal DataArray where each day takes the value defined by the day-of-year.

Parameters

- **doy** (*xr.DataArray*) – Array with *dayofyear* coordinate.
- **arr** (*xr.DataArray or xr.Dataset*) – Array with *time* coordinate.

Returns

xr.DataArray – An array with the same dimensions as *doy*, except for *dayofyear*, which is replaced by the *time* dimension of *arr*. Values are filled according to the day of year value in *doy*.

`xclim.core.calendar.select_time(`*da: xr.DataArray | xr.Dataset, drop: bool = False, season: str | Sequence[str] = None, month: int | Sequence[int] = None, doy_bounds: tuple[int, int] = None, date_bounds: tuple[str, str] = None*`)` → xr.DataArray | xr.Dataset

Select entries according to a time period.

This conveniently improves xarray's `xarray.DataArray.where()` and `xarray.DataArray.sel()` with fancier ways of indexing over time elements. In addition to the data *da* and argument *drop*, only one of *season*, *month*, *doy_bounds* or *date_bounds* may be passed.

Parameters

- **da** (*xr.DataArray or xr.Dataset*) – Input data.
- **drop** (*boolean*) – Whether to drop elements outside the period of interest or to simply mask them (default).
- **season** (*string or sequence of strings*) – One or more of 'DJF', 'MAM', 'JJA' and 'SON'.
- **month** (*integer or sequence of integers*) – Sequence of month numbers (January = 1 ... December = 12)
- **doy_bounds** (*2-tuple of integers*) – The bounds as (start, end) of the period of interest expressed in day-of-year, integers going from 1 (January 1st) to 365 or 366 (December 31st). If calendar awareness is needed, consider using `date_bounds` instead. Bounds are inclusive.
- **date_bounds** (*2-tuple of strings*) – The bounds as (start, end) of the period of interest expressed as dates in the month-day (%m-%d) format. Bounds are inclusive.

Returns

xr.DataArray or xr.Dataset – Selected input values. If `drop=False`, this has the same length as `da` (along dimension 'time'), but with masked (NaN) values outside the period of interest.

Examples

Keep only the values of fall and spring.

```
>>> ds = open_dataset("ERA5/daily_surface_cancities_1990-1993.nc")
>>> ds.time.size
1461
>>> out = select_time(ds, drop=True, season=["MAM", "SON"])
>>> out.time.size
732
```

Or all values between two dates (included).

```
>>> out = select_time(ds, drop=True, date_bounds=("02-29", "03-02"))
>>> out.time.values
array(['1990-03-01T00:00:00.000000000', '1990-03-02T00:00:00.000000000',
       '1991-03-01T00:00:00.000000000', '1991-03-02T00:00:00.000000000',
       '1992-02-29T00:00:00.000000000', '1992-03-01T00:00:00.000000000',
       '1992-03-02T00:00:00.000000000', '1993-03-01T00:00:00.000000000',
       '1993-03-02T00:00:00.000000000'], dtype='datetime64[ns]')
```

`xclim.core.calendar.time_bnds`(*group*, *freq: str*) → Sequence[tuple[cftime.datetime, cftime.datetime]]

Find the time bounds for a pseudo-period index.

As we are using datetime indices to stand in for period indices, assumptions regarding the period are made based on the given freq. IMPORTANT NOTE: this function cannot be used on greater-than-day freq that start at the beginning of a month, e.g. 'MS', 'QS', 'AS' – this mirrors pandas behavior.

Parameters

- **group** (*CFTimeIndex or DataArrayResample*) – Object which contains CFTimeIndex as a proxy representation for CFPeriodIndex
- **freq** (*str*) – String specifying the frequency/offset such as 'MS', '2D', or '3T'

Returns

Sequence[(cftime.datetime, cftime.datetime)] – The start and end times of the period inferred from datetime and freq.

Examples

```
>>> from xarray import cftime_range
>>> from xclim.core.calendar import time_bnds
>>> index = cftime_range(
...     start="2000-01-01", periods=3, freq="2QS", calendar="360_day"
... )
>>> out = time_bnds(index, "2Q")
>>> for bnds in out:
...     print(
...         bnds[0].strftime("%Y-%m-%dT%H:%M:%S"),
...         " -",
...         bnds[1].strftime("%Y-%m-%dT%H:%M:%S"),
...     )
...
2000-01-01T00:00:00  - 2000-03-30T23:59:59
2000-07-01T00:00:00  - 2000-09-30T23:59:59
2001-01-01T00:00:00  - 2001-03-30T23:59:59
```

`xclim.core.calendar.within_bnds_doy`(*arr: DataArray*, ***, *low: DataArray*, *high: DataArray*) → DataArray

Return whether or not array values are within bounds for each day of the year.

Parameters

- **arr** (*xarray.DataArray*) – Input array.
- **low** (*xarray.DataArray*) – Low bound with dayofyear coordinate.
- **high** (*xarray.DataArray*) – High bound with dayofyear coordinate.

Returns

xarray.DataArray

14.6.2 Formatting utilities for indicators

class `xclim.core.formatting.AttrFormatter`(*mapping: Mapping[str, Sequence[str]]*, *modifiers: Sequence[str]*)

Bases: `Formatter`

A formatter for frequently used attribute values.

See the doc of format_field() for more details.

`format(format_string: str, /, *args: Any, **kwargs: dict) → str`

Format a string.

Parameters

- **format_string** (*str*)
- **args**
- **kwargs**

Returns

str

`format_field(value, format_spec)`

Format a value given a formatting spec.

If *format_spec* is in this Formatter's modifiers, the corresponding variation of value is given. If *format_spec* is 'r' (raw), the value is returned unmodified. If *format_spec* is not specified but *value* is in the mapping, the first variation is returned.

Examples

Let's say the string "The dog is {adj1}, the goose is {adj2}" is to be translated to french and that we know that possible values of *adj* are *nice* and *evil*. In french, the genre of the noun changes the adjective (cat = chat is masculine, and goose = oie is feminine) so we initialize the formatter as:

```
>>> fmt = AttrFormatter(
...     {
...         "nice": ["beau", "belle"],
...         "evil": ["méchant", "méchante"],
...         "smart": ["intelligent", "intelligente"],
...     },
...     ["m", "f"],
... )
>>> fmt.format(
...     "Le chien est {adj1:m}, l'oie est {adj2:f}, le gecko est {adj3:r}",
...     adj1="nice",
...     adj2="evil",
...     adj3="smart",
... )
"Le chien est beau, l'oie est méchante, le gecko est smart"
```

The base values may be given using unix shell-like patterns:

```
>>> fmt = AttrFormatter(
...     {"AS-*": ["annuel", "annuelle"], "MS": ["mensuel", "mensuelle"]},
...     ["m", "f"],
... )
>>> fmt.format(
...     "La moyenne {freq:f} est faite sur un échantillon {src_timestep:m}",
...     freq="AS-JUL",
...     src_timestep="MS",
... )
'La moyenne annuelle est faite sur un échantillon mensuel'
```

`xclim.core.formatting.gen_call_string`(*funcname: str*, **args*, ***kwargs*)

Generate a signature string for use in the history attribute.

DataArrays and Dataset are replaced with their name, while Nones, floats, ints and strings are printed directly. All other objects have their type printed between < >.

Arguments given through positional arguments are printed positionnally and those given through keywords are printed prefixed by their name.

Parameters

- **funcname** (*str*) – Name of the function
- **args, kwargs** – Arguments given to the function.

Example

```
>>> A = xr.DataArray([1], dims=("x",), name="A")
>>> gen_call_string("func", A, b=2.0, c="3", d=[4, 5, 6])
"func(A, b=2.0, c='3', d=<list>)"
```

`xclim.core.formatting.generate_indicator_docstring`(*ind*)

Generate an indicator's docstring from keywords.

Parameters

ind (*Indicator instance*)

`xclim.core.formatting.get_percentile_metadata`(*data: xr.DataArray*, *prefix: str*) → dict[str, str]

Get the metadata related to percentiles from the given DataArray as a dictionary.

Parameters

- **data** (*xr.DataArray*) – Must be compatible with PercentileDataArray, this means the necessary metadata must be available in its attributes and coordinates.
- **prefix** (*str*) – The prefix to be used in the metadata key. Usually this takes the form of "tasmin_per" or equivalent.

Returns

dict – A mapping of the configuration used to compute these percentiles.

`xclim.core.formatting.merge_attributes`(*attribute: str*, **inputs_list: xr.DataArray | xr.Dataset*, *new_line: str = '\n'*, *missing_str: str | None = None*, ***inputs_kws: xr.DataArray | xr.Dataset*)

Merge attributes from several DataArrays or Datasets.

If more than one input is given, its name (if available) is prepended as: "<input name> : <input attribute>".

Parameters

- **attribute** (*str*) – The attribute to merge.
- **inputs_list** (*Union[xr.DataArray, xr.Dataset]*) – The datasets or variables that were used to produce the new object. Inputs given that way will be prefixed by their *name* attribute if available.
- **new_line** (*str*) – The character to put between each instance of the attributes. Usually, in CF-conventions, the history attributes uses '\n' while cell_methods uses ' '.

- **missing_str** (*str*) – A string that is printed if an input doesn't have the attribute. Defaults to None, in which case the input is simply skipped.
- **inputs_kws** (*Union[xr.DataArray, xr.Dataset]*) – Mapping from names to the datasets or variables that were used to produce the new object. Inputs given that way will be prefixes by the passed name.

Returns
str – The new attribute made from the combination of the ones from all the inputs.

`xclim.core.formatting.parse_doc(`*doc: str*`)` → dict[str, str]

Crude regex parsing reading an indice docstring and extracting information needed in indicator construction.

The appropriate docstring syntax is detailed in *Defining new indices*.

Parameters
doc (*str*) – The docstring of an indice function.

Returns
dict – A dictionary with all parsed sections.

`xclim.core.formatting.prefix_attrs(`*source: dict*, *keys: Sequence*, *prefix: str*`)`

Rename some keys of a dictionary by adding a prefix.

Parameters

- **source** (*dict*) – Source dictionary, for example data attributes.
- **keys** (*sequence*) – Names of keys to prefix.
- **prefix** (*str*) – Prefix to prepend to keys.

Returns
dict – Dictionary of attributes with some keys prefixed.

`xclim.core.formatting.unprefix_attrs(`*source: dict*, *keys: Sequence*, *prefix: str*`)`

Remove prefix from keys in a dictionary.

Parameters

- **source** (*dict*) – Source dictionary, for example data attributes.
- **keys** (*sequence*) – Names of original keys for which prefix should be removed.
- **prefix** (*str*) – Prefix to remove from keys.

Returns
dict – Dictionary of attributes whose keys were prefixed, with prefix removed.

`xclim.core.formatting.update_history(`*hist_str: str*, **inputs_list: Sequence[xr.DataArray | xr.Dataset]*, *new_name: str | None = None*, ***inputs_kws: Mapping[str, xr.DataArray | xr.Dataset]*`)`

Return a history string with the timestamped message and the combination of the history of all inputs.

The new history entry is formatted as "[<timestamp>] <new_name>: <hist_str> - xclim version: <xclim.__version__>."

Parameters

- **hist_str** (*str*) – The string describing what has been done on the data.
- **new_name** (*Optional[str]*) – The name of the newly created variable or dataset to prefix hist_msg.

- **inputs_list** (*Sequence[Union[xr.DataArray, xr.Dataset]]*) – The datasets or variables that were used to produce the new object. Inputs given that way will be prefixed by their "name" attribute if available.
- **inputs_kws** (*Mapping[str, Union[xr.DataArray, xr.Dataset]]*) – Mapping from names to the datasets or variables that were used to produce the new object. Inputs given that way will be prefixes by the passed name.

Returns
str – The combine history of all inputs starting with *hist_str*.

See also:

merge_attributes

`xclim.core.formatting.update_xclim_history(`*func*`)`

Decorator that auto-generates and fills the history attribute.

The history is generated from the signature of the function and added to the first output. Because of a limitation of the *boltons* wrapper, all arguments passed to the wrapped function will be printed as keyword arguments.

14.6.3 Options submodule

Global or contextual options for xclim, similar to xarray.set_options.

`class xclim.core.options.set_options(`***kwargs*`)`

Set options for xclim in a controlled context.

Currently-supported options:

- `metadata_locales`: List of IETF language tags or tuples of language tags and a translation dict, or tuples of language tags and a path to a json file defining translation of attributes. Default: `[]`.
- `data_validation`: Whether to 'log', 'raise' an error or 'warn' the user on inputs that fail the data checks in *xclim.core.datachecks*. Default: `'raise'`.
- `cf_compliance`: Whether to 'log', 'raise' an error or 'warn' the user on inputs that fail the CF compliance checks in *xclim.core.cfchecks*. Default: `'warn'`.
- `check_missing`: How to check for missing data and flag computed indicators. Default available methods are "any", "wmo", "pct", "at_least_n" and "skip". Missing method can be registered through the *xclim.core.options.register_missing_method* decorator. Default: `'any'`
- `missing_options`: Dictionary of options to pass to the missing method. Keys must the name of missing method and values must be mappings from option names to values.
- `run_length_ufunc`: Whether to use the 1D ufunc version of run length algorithms or the dask-ready broadcasting version. Default is `'auto'` which means the latter is used for dask-backed and large arrays.
- `sdba_extra_output`: Whether to add diagnostic variables to outputs of sdba's *train*, *adjust* and *processing* operations. Details about these additional variables are given in the object's docstring. When activated, *adjust* will return a Dataset with *scen* and those extra diagnostics For *processing* functions, see the doc, the output type might change, or not depending on the algorithm. Default: `False`.
- `sdba_encode_cf`: Whether to encode cf coordinates in the `map_blocks` optimization that most adjustment methods are based on. This should have no impact on the results, but should run much faster in the graph creation phase.

- `keep_attrs`: Controls attributes handling in indicators. If True, attributes from all inputs are merged using the *drop_conflicts* strategy and then updated with xclim-provided attributes. If False, attributes from the inputs are ignored. If "xarray", xclim will use xarray's *keep_attrs* option. Note that xarray's "default" is equivalent to False. Default: `"xarray"`.

Examples

You can use `set_options` either as a context manager:

```
>>> import xclim
>>> ds = xr.open_dataset(path_to_tas_file).tas
>>> with xclim.set_options(metadata_locales=["fr"]):
...     out = xclim.atmos.tg_mean(ds)
...
```

Or to set global options:

```
>>> xclim.set_options(
...     missing_options={"pct": {"tolerance": 0.04}}
... )
<xclim.core.options.set_options object at ...>
```

14.6.4 Miscellaneous indices utilities

Helper functions for the indices computations, indicator construction and other things.

`xclim.core.utils.DateStr`

Type annotation for strings representing full dates (YYYY-MM-DD), may include time.

alias of `str`

`xclim.core.utils.DayOfYearStr`

Type annotation for strings representing dates without a year (MM-DD).

alias of `str`

`xclim.core.utils.wrapped_partial`(*func: FunctionType*, *suggested: dict | None = None*, ***fixed*) → Callable

Wrap a function, updating its signature but keeping its docstring.

Parameters

- **func** (*FunctionType*) – The function to be wrapped
- **suggested** (*dict*) – Keyword arguments that should have new default values but still appear in the signature.
- **fixed** (*kwargs*) – Keyword arguments that should be fixed by the wrapped and removed from the signature.

Examples

```
>>> from inspect import signature
>>> def func(a, b=1, c=1):
...     print(a, b, c)
...
>>> newf = wrapped_partial(func, b=2)
>>> signature(newf)
<Signature (a, *, c=1)>
>>> newf(1)
1 2 1
>>> newf = wrapped_partial(func, suggested=dict(c=2), b=2)
>>> signature(newf)
<Signature (a, *, c=2)>
>>> newf(1)
1 2 2
```

`xclim.core.utils.walk_map(`*d: dict*, *func: function*`)` → dict

Apply a function recursively to values of dictionary.

Parameters

- **d** (*dict*) – Input dictionary, possibly nested.
- **func** (*FunctionType*) – Function to apply to dictionary values.

Returns

dict – Dictionary whose values are the output of the given function.

`xclim.core.utils.load_module(`*path: os.PathLike*, *name: str | None = None*`)`

Load a python module from a python file, optionally changing its name.

Examples

Given a path to a module file (.py)

```
>>> # xdoctest: +SKIP
>>> from pathlib import Path
>>> path = Path("path/to/example.py")
```

The two following imports are equivalent, the second uses this method.

```
>>> os.chdir(path.parent)
>>> import example as mod1
>>> os.chdir(previous_working_dir)
>>> mod2 = load_module(path)
>>> mod1 == mod2
```

`exception xclim.core.utils.ValidationError`

Bases: `ValueError`

Error raised when input data to an indicator fails the validation tests.

`property msg`

`exception xclim.core.utils.MissingVariableError`

Bases: `ValueError`

Error raised when a dataset is passed to an indicator but one of the needed variable is missing.

`xclim.core.utils.ensure_chunk_size(`*da: DataArray*, ***minchunks: Mapping[str, int]*`)` → DataArray

Ensure that the input DataArray has chunks of at least the given size.

If only one chunk is too small, it is merged with an adjacent chunk. If many chunks are too small, they are grouped together by merging adjacent chunks.

Parameters

- **da** (*xr.DataArray*) – The input DataArray, with or without the dask backend. Does nothing when passed a non-dask array.
- **minchunks** (*Mapping[str, int]*) – A kwarg mapping from dimension name to minimum chunk size. Pass -1 to force a single chunk along that dimension.

`xclim.core.utils.uses_dask(`*da*`)`

Evaluate whether dask is installed and array is loaded as a dask array.

`xclim.core.utils.calc_perc(`*arr: ndarray*, *percentiles: Optional[Sequence[float]] = None*, *alpha: float = 1.0*, *beta: float = 1.0*, *copy: bool = True*`)` → ndarray

Compute percentiles using nan_calc_percentiles and move the percentiles' axis to the end.

`xclim.core.utils.nan_calc_percentiles(`*arr: ndarray*, *percentiles: Optional[Sequence[float]] = None*, *axis=- 1*, *alpha=1.0*, *beta=1.0*, *copy=True*`)` → ndarray

Convert the percentiles to quantiles and compute them using _nan_quantile.

`xclim.core.utils.raise_warn_or_log(`*err: Exception*, *mode: str*, *msg: str | None = None*, *err_type=<class 'ValueError'>*, *stacklevel: int = 1*`)`

Raise, warn or log an error according.

Parameters

- **err** (*Exception*) – An error.
- **mode** (*{'ignore', 'log', 'warn', 'raise'}*) – What to do with the error.
- **msg** (*str, optional*) – The string used when logging or warning. Defaults to the *msg* attr of the error (if present) or to "Failed with <err>".
- **err_type** (*type*) – The type of error/exception to raise.
- **stacklevel** (*int*) – Stacklevel when warning. Relative to the call of this function (1 is added).

`class xclim.core.utils.InputKind(`*value*`)`

Bases: `IntEnum`

Constants for input parameter kinds.

For use by external parses to determine what kind of data the indicator expects. On the creation of an indicator, the appropriate constant is stored in `xclim.core.indicator.Indicator.parameters`. The integer value is what gets stored in the output of `xclim.core.indicator.Indicator.json()`.

For developers : for each constant, the docstring specifies the annotation a parameter of an indice function should use in order to be picked up by the indicator constructor. Notice that we are using the annotation format as described in PEP604/py3.10, i.e. with | indicating an union and without import objects from *typing*.

`VARIABLE = 0`

A data variable (DataArray or variable name).

Annotation : `xr.DataArray`.

`OPTIONAL_VARIABLE = 1`

An optional data variable (DataArray or variable name).

Annotation : `xr.DataArray | None`. The default should be None.

`QUANTITY_STR = 2`

A string representing a quantity with units.

Annotation : `str` + an entry in the *`xclim.core.units.declare_units()`* decorator.

`FREQ_STR = 3`

A string representing an “offset alias”, as defined by pandas.

See https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases . Annotation : `str` + `freq` as the parameter name.

`NUMBER = 4`

A number.

Annotation : `int`, `float` and unions thereof, potentially optional.

`STRING = 5`

A simple string.

Annotation : `str` or `str | None`. In most cases, this kind of parameter makes sense with choices indicated in the docstring’s version of the annotation with curly braces. See *Defining new indices*.

`DAY_OF_YEAR = 6`

A date, but without a year, in the MM-DD format.

Annotation : *`xclim.core.utils.DayOfYearStr`* (may be optional).

`DATE = 7`

A date in the YYYY-MM-DD format, may include a time.

Annotation : *`xclim.core.utils.DateStr`* (may be optional).

`NUMBER_SEQUENCE = 8`

A sequence of numbers

Annotation : `Sequence[int]`, `Sequence[float]` and unions thereof, may include single `int` and `float`, may be optional.

`BOOL = 9`

A boolean flag.

Annotation : `bool`, may be optional.

`KWARGS = 50`

A mapping from argument name to value.

Developers : maps the `**kwargs`. Please use as little as possible.

`DATASET = 70`

An xarray dataset.

Developers : as indices only accept DataArrays, this should only be added on the indicator’s constructor.

`OTHER_PARAMETER = 99`

An object that fits None of the previous kinds.

Developers : This is the fallback kind, it will raise an error in xclim's unit tests if used.

`xclim.core.utils.infer_kind_from_parameter`(*param: Parameter*, *has_units: bool = False*) → *InputKind*

Return the appropriate InputKind constant from an `inspect.Parameter` object.

The correspondance between parameters and kinds is documented in `xclim.core.utils.InputKind`. The only information not inferable through the *inspect* object is whether the parameter has been assigned units through the `xclim.core.units.declare_units()` decorator. That can be given with the `has_units` flag.

`xclim.core.utils.adapt_clix_meta_yaml`(*raw: PathLike*, *adapted: PathLike*)

Read in a clix-meta yaml and refactor it to fit xclim's yaml specifications.

`class xclim.core.utils.PercentileDataArray`(*data: Any = <NA>*, *coords: Sequence[tuple] | Mapping[Any, Any] | None = None*, *dims: Hashable | Sequence[Hashable] | None = None*, *name: Hashable = None*, *attrs: Mapping = None*, *indexes: dict[Hashable, pd.Index] = None*, *fastpath: bool = False*)

Bases: `DataArray`

Wrap xarray DataArray for percentiles values.

This class is used internally with its corresponding InputKind to recognize this sort of input and to retrieve from it the attributes needed to build indicator metadata.

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

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

Parameters

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

Returns

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

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

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

Parameters

- **dim** (*str or sequence of str, optional*) – Dimension(s) over which to apply *any*.
- **axis** (*int or sequence of int, optional*) – Axis(es) over which to apply *any*. Only one of the 'dim' and 'axis' arguments can be supplied. If neither are supplied, then *any* is calculated over axes.

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

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

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

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

Parameters

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

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

`cumprod(`*dim=None*, *axis=None*, *skipna=None*, ***kwargs*`)`

Apply *cumprod* along some dimension of PercentileDataArray.

Parameters

- **dim** (*str or sequence of str, optional*) – Dimension over which to apply *cumprod*.
- **axis** (*int or sequence of int, optional*) – Axis over which to apply *cumprod*. Only one of the 'dim' and 'axis' arguments can be supplied.
- **skipna** (*bool, optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **keep_attrs** (*bool, optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- ****kwargs** (*dict*) – Additional keyword arguments passed on to *cumprod*.

Returns
cumvalue (*PercentileDataArray*) – New PercentileDataArray object with *cumprod* applied to its data along the indicated dimension.

`cumsum(`*dim=None*, *axis=None*, *skipna=None*, ***kwargs*`)`

Apply *cumsum* along some dimension of PercentileDataArray.

Parameters

- **dim** (*str or sequence of str, optional*) – Dimension over which to apply *cumsum*.
- **axis** (*int or sequence of int, optional*) – Axis over which to apply *cumsum*. Only one of the 'dim' and 'axis' arguments can be supplied.
- **skipna** (*bool, optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **keep_attrs** (*bool, optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- ****kwargs** (*dict*) – Additional keyword arguments passed on to *cumsum*.

Returns
cumvalue (*PercentileDataArray*) – New PercentileDataArray object with *cumsum* applied to its data along the indicated dimension.

`item(*args)`

Copy an element of an array to a standard Python scalar and return it.

Parameters
***args** (*Arguments (variable number and type)*) –

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns
z (*Standard Python scalar object*) – A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

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

Reduce this PercentileDataArray's data by applying *max* along some dimension(s).

Parameters

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

Returns

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

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

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

Parameters

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

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

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

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

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

Parameters

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

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

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

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

Parameters

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

Returns

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

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

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

Parameters

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

Returns

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

`searchsorted`(*v*, *side='left'*, *sorter=None*)

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see *numpy.searchsorted*

See also:

`numpy.searchsorted`
equivalent function

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

Reduce this PercentileDataArray's data by applying *std* along some dimension(s).

Parameters

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

a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).

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

Returns

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

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

Reduce this PercentileDataArray's data by applying *sum* along some dimension(s).

Parameters

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

Returns

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

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

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

Parameters

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

a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).

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

Returns

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

`classmethod is_compatible(`*source: DataArray*`)` → bool

Evaluate whether PecentileDataArray is conformant with expected fields.

A PercentileDataArray must have climatology_bounds attributes and either a quantile or percentiles coordinate, the window is not mandatory.

`classmethod from_da(`*source: xr.DataArray*, *climatology_bounds: list[str] = None*`)` → *PercentileDataArray*

Create a PercentileDataArray from a xarray.DataArray.

Parameters

- **source** (*DataArray*) – A DataArray with its content containing percentiles values. It must also have a coordinate variable percentiles or quantile.
- **climatology_bounds** (*list[str]*) – Optional. A List of size two which contains the period on which the percentiles were computed. See *xclim.core.calendar.build_climatology_bounds* to build this list from a DataArray.

Returns

PercentileDataArray – The initial *source* DataArray but wrap by PercentileDataArray class. The data is unchanged and only climatology_bounds attributes is overridden if q new value is given in inputs.

14.7 Other xclim modules

14.7.1 Spatial Analogs module

See *Spatial Analogues*.

14.7.2 Testing module

Helpers for testing xclim.

Testing and tutorial utilities' module.

`xclim.testing.utils.get_all_CMIP6_variables(`*get_cell_methods=True*`)`

`xclim.testing.utils.list_datasets(`*github_repo='Ouranosinc/xclim-testdata'*, *branch='main'*`)`

Return a DataFrame listing all xclim test datasets available on the GitHub repo for the given branch.

The result includes the filepath, as passed to *open_dataset*, the file size (in KB) and the html url to the file. This uses an unauthenticated call to GitHub's REST API, so it is limited to 60 requests per hour (per IP). A single call of this function triggers one request per subdirectory, so use with parsimony.

`xclim.testing.utils.list_input_variables(`*submodules: Optional[Sequence[str]] = None*, *realms: Optional[Sequence[str]] = None*`)` → dict

List all possible variables names used in xclim's indicators.

Made for development purposes. Parses all indicator parameters with the `xclim.core.utils.InputKind.VARIABLE` or *OPTIONAL_VARIABLE* kinds.

Parameters

- **realms** (*Sequence of str, optional*) – Restrict the output to indicators of a list of realms only. Default None, which parses all indicators.
- **submodules** (*str, optional*) – Restrict the output to indicators of a list of submodules only. Default None, which parses all indicators.

Returns

dict – A mapping from variable name to indicator class.

`xclim.testing.utils.open_dataset(`*name: str*, *suffix: str | None = None*, *dap_url: str | None = None*, *github_url: str = 'https://github.com/Ouranosinc/xclim-testdata'*, *branch: str = 'main'*, *cache: bool = True*, *cache_dir: Path = PosixPath('/home/docs/.xclim_testing_data')*, ***kwargs*`)` → Dataset

Open a dataset from the online GitHub-like repository.

If a local copy is found then always use that to avoid network traffic.

Parameters

- **name** (*str*) – Name of the file containing the dataset.
- **suffix** (*str, optional*) – If no suffix is given, assumed to be netCDF ('.nc' is appended). For no suffix, set "".
- **dap_url** (*str, optional*) – URL to OPeNDAP folder where the data is stored. If supplied, supersedes github_url.
- **github_url** (*str*) – URL to GitHub repository where the data is stored.
- **branch** (*str, optional*) – For GitHub-hosted files, the branch to download from.
- **cache_dir** (*Path*) – The directory in which to search for and write cached data.
- **cache** (*bool*) – If True, then cache data locally for use on subsequent calls.
- **kwargs** – For NetCDF files, keywords passed to `xarray.open_dataset()`.

Returns

Union[Dataset, Path]

See also:

`xarray.open_dataset`

`xclim.testing.utils.publish_release_notes(`*style: str = 'md'*, *file: os.PathLike | StringIO | TextIO | None = None*`)` → str | None

Format release history in Markdown or ReStructuredText.

Parameters

- **style** (*{"rst", "md"}*) – Use ReStructuredText formatting or Markdown. Default: Markdown.
- **file** (*{os.PathLike, StringIO, TextIO}, optional*) – If provided, prints to the given file-like object. Otherwise, returns a string.

Returns
str, optional

Notes

This function is solely for development purposes.

`xclim.testing.utils.show_versions`(*file: os.PathLike | StringIO | TextIO | None = None*) → str | None

Print the versions of xclim and its dependencies.

Parameters
file (*{os.PathLike, StringIO, TextIO}, optional*) – If provided, prints to the given file-like object. Otherwise, returns a string.

Returns
str or None

`xclim.testing.utils.update_variable_yaml`(*filename=None, xclim_needs_only=True*)

Update a variable from a yaml file.

14.7.3 Subset module

Warning: Subsetting is now offered via *clisops.core.subset*. The subsetting functions offered by *clisops* are available at the following link:

CLISOPS API

Note: For more information about *clisops* refer to their documentation here: CLISOPS documentation

CHAPTER

FIFTEEN

XCLIM

15.1 xclim package

15.1.1 Subpackages

xclim.core package

Core module.

Submodules

xclim.core.bootstrapping module

Module comprising the bootstrapping algorithm for indicators.

`xclim.core.bootstrapping._get_bootstrap_freq(`*freq*`)`

`xclim.core.bootstrapping._get_year_label(`*year_dt*`)` → str

`xclim.core.bootstrapping.bootstrap_func(`*compute_index_func: Callable*, ***kwargs*`)` → DataArray

Bootstrap the computation of percentile-based exceedance indices.

Indices measuring exceedance over percentile-based threshold may contain artificial discontinuities at the beginning and end of the reference period used for calculating the percentile. A bootstrap resampling procedure can reduce those discontinuities by iteratively replacing each the year the indice is computed on from the percentile estimate, and replacing it with another year within the reference period.

Parameters

- **compute_index_func** (*Callable*) – Indice function.
- **kwargs** (*dict*) – Arguments to *func*.

Returns

xr.DataArray – The result of func with bootstrapping.

References

Zhang, X., Hegerl, G., Zwiers, F. W., & Kenyon, J. (2005). Avoiding Inhomogeneity in Percentile-Based Indices of Temperature Extremes, Journal of Climate, 18(11), 1641-1651, https://doi.org/10.1175/JCLI3366.1

Notes

This function is meant to be used by the *percentile_bootstrap* decorator. The parameters of the percentile calculation (percentile, window, reference_period) are stored in the attributes of the percentile DataArray. The bootstrap algorithm implemented here does the following:

```
For each temporal grouping in the calculation of the indice
    If the group `g_t` is in the reference period
        For every other group `g_s` in the reference period
            Replace group `g_t` by `g_s`
            Compute percentile on resampled time series
            Compute indice function using percentile
        Average output from indice function over all resampled time series
    Else compute indice function using original percentile
```

`xclim.core.bootstrapping.build_bootstrap_year_da`(*da: DataArray*, *groups: dict[Any, slice]*, *label: Any*, *dim: str = 'time'*) → DataArray

Return an array where a group in the original is replaced by every other groups along a new dimension.

Parameters

- **da** (*DataArray*) – Original input array over reference period.
- **groups** (*dict*) – Output of grouping functions, such as *DataArrayResample.groups*.
- **label** (*Any*) – Key identifying the group item to replace.
- **dim** (*str*) – Dimension recognized as time. Default: *time*.

Returns

DataArray – Array where one group is replaced by values from every other group along the *bootstrap* dimension.

`xclim.core.bootstrapping.percentile_bootstrap`(*func*)

Decorator applying a bootstrap step to the calculation of exceedance over a percentile threshold.

This feature is experimental.

Bootstraping avoids discontinuities in the exceedance between the reference period over which percentiles are computed, and "out of reference" periods. See *bootstrap_func* for details.

Example of declaration:

```
>>>
>>> @declare_units(tas="[temperature]", t90="[temperature]")
```

… @percentile_bootstrap … def tg90p(… tas: xarray.DataArray, … t90: xarray.DataArray, … freq: str = "YS", … bootstrap: bool = False, …) -> xarray.DataArray: … pass

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tg90p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> # To start bootstrap reference period must not fully overlap the studied period.
>>> tas_ref = tas.sel(time=slice("1990-01-01", "1992-12-31"))
>>> t90 = percentile_doy(tas_ref, window=5, per=90)
>>> tg90p(tas=tas, tas_per=t90.sel(percentiles=90), freq="YS", bootstrap=True)
```

xclim.core.calendar module

Calendar handling utilities

Helper function to handle dates, times and different calendars with xarray.

`xclim.core.calendar.DayOfYearStr`

Type annotation for strings representing dates without a year (MM-DD).

alias of `str`

`xclim.core.calendar.adjust_doy_calendar`(*source: xr.DataArray*, *target: xr.DataArray | xr.Dataset*) → xr.DataArray

Interpolate from one set of dayofyear range to another calendar.

Interpolate an array defined over a *dayofyear* range (say 1 to 360) to another *dayofyear* range (say 1 to 365).

Parameters

- **source** (*xr.DataArray*) – Array with *dayofyear* coordinate.
- **target** (*xr.DataArray or xr.Dataset*) – Array with *time* coordinate.

Returns

xr.DataArray – Interpolated source array over coordinates spanning the target *dayofyear* range.

`xclim.core.calendar.cfindex_end_time`(*cfindex: CFTimeIndex*, *freq: str*) → CFTimeIndex

Get the end of a period for a pseudo-period index.

As we are using datetime indices to stand in for period indices, assumptions regarding the period are made based on the given freq. IMPORTANT NOTE: this function cannot be used on greater-than-day freq that start at the beginning of a month, e.g. ‘MS’, ‘QS’, ‘AS’ – this mirrors pandas behavior.

Parameters

- **cfindex** (*CFTimeIndex*) – CFTimeIndex as a proxy representation for CFPeriodIndex
- **freq** (*str*) – String specifying the frequency/offset such as ‘MS’, ‘2D’, ‘H’, or ‘3T’

Returns

CFTimeIndex – The ending datetimes of periods inferred from dates and freq `xclim.core.calendar.cfindex_start_time(`*cfindex: CFTimeIndex*, *freq: str*`)` → CFTimeIndex

Get the start of a period for a pseudo-period index.

As we are using datetime indices to stand in for period indices, assumptions regarding the period are made based on the given freq. IMPORTANT NOTE: this function cannot be used on greater-than-day freq that start at the beginning of a month, e.g. 'MS', 'QS', 'AS' – this mirrors pandas behavior.

Parameters

- **cfindex** (*CFTimeIndex*) – CFTimeIndex as a proxy representation for CFPeriodIndex
- **freq** (*str*) – String specifying the frequency/offset such as 'MS', '2D', 'H', or '3T'

Returns

CFTimeIndex – The starting datetimes of periods inferred from dates and freq

`xclim.core.calendar.cftime_end_time(`*date: datetime*, *freq: str*`)` → datetime

Get the cftime.datetime for the end of a period.

As we are not supplying actual period objects, assumptions regarding the period are made based on the given freq. IMPORTANT NOTE: this function cannot be used on greater-than-day freq that start at the beginning of a month, e.g. 'MS', 'QS', 'AS' – this mirrors pandas behavior.

Parameters

- **date** (*cftime.datetime*) – The original datetime object as a proxy representation for period.
- **freq** (*str*) – String specifying the frequency/offset such as 'MS', '2D', 'H', or '3T'

Returns

cftime.datetime – The ending datetime of the period inferred from date and freq.

`xclim.core.calendar.cftime_start_time(`*date: datetime*, *freq: str*`)` → datetime

Get the cftime.datetime for the start of a period.

As we are not supplying actual period objects, assumptions regarding the period are made based on the given freq. IMPORTANT NOTE: this function cannot be used on greater-than-day freq that start at the beginning of a month, e.g. 'MS', 'QS', 'AS' – this mirrors pandas behavior.

Parameters

- **date** (*cftime.datetime*) – The original datetime object as a proxy representation for period.
- **freq** (*str*) – String specifying the frequency/offset such as 'MS', '2D', 'H', or '3T'

Returns

cftime.datetime – The starting datetime of the period inferred from date and freq.

`xclim.core.calendar.climatological_mean_doy(`*arr: xr.DataArray*, *window: int = 5*`)` → tuple[xr.DataArray, xr.DataArray]

Calculate the climatological mean and standard deviation for each day of the year.

Parameters

- **arr** (*xarray.DataArray*) – Input array.
- **window** (*int*) – Window size in days.

Returns

xarray.DataArray, xarray.DataArray – Mean and standard deviation.

`xclim.core.calendar.compare_offsets`(*freqA: str*, *op: str*, *freqB: str*) → bool

Compare offsets string based on their approximate length, according to a given operator.

Offset are compared based on their length approximated for a period starting after 1970-01-01 00:00:00. If the offsets are from the same category (same first letter), only the multiplicator prefix is compared (QS-DEC == QS-JAN, MS < 2MS). "Business" offsets are not implemented.

Parameters

- **freqA** (*str*) – RHS Date offset string ('YS', '1D', 'QS-DEC', . . .)
- **op** (*{'<', '<=', '==', '>', '>=', '!='}*) – Operator to use.
- **freqB** (*str*) – LHS Date offset string ('YS', '1D', 'QS-DEC', . . .)

Returns

bool – freqA op freqB

`xclim.core.calendar.convert_calendar`(*source: xr.DataArray | xr.Dataset*, *target: xr.DataArray | str*, *align_on: str | None = None*, *missing: Any | None = None*, *dim: str = 'time'*) → xr.DataArray | xr.Dataset

Convert a DataArray/Dataset to another calendar using the specified method.

Only converts the individual timestamps, does not modify any data except in dropping invalid/surplus dates or inserting missing dates.

If the source and target calendars are either no_leap, all_leap or a standard type, only the type of the time array is modified. When converting to a leap year from a non-leap year, the 29th of February is removed from the array. In the other direction and if *target* is a string, the 29th of February will be missing in the output, unless *missing* is specified, in which case that value is inserted.

For conversions involving *360_day* calendars, see Notes.

This method is safe to use with sub-daily data as it doesn't touch the time part of the timestamps.

Parameters

- **source** (*xr.DataArray*) – Input array/dataset with a time coordinate of a valid dtype (datetime64 or a cftime.datetime).
- **target** (*Union[xr.DataArray, str]*) – Either a calendar name or the 1D time coordinate to convert to. If an array is provided, the output will be reindexed using it and in that case, days in *target* that are missing in the converted *source* are filled by *missing* (which defaults to NaN).
- **align_on** (*{None, 'date', 'year', 'random'}*) – Must be specified when either source or target is a *360_day* calendar, ignored otherwise. See Notes.
- **missing** (*Optional[any]*) – A value to use for filling in dates in the target that were missing in the source. If *target* is a string, default (None) is not to fill values. If it is an array, default is to fill with NaN.
- **dim** (*str*) – Name of the time coordinate.

Returns

Union[xr.DataArray, xr.Dataset] – Copy of source with the time coordinate converted to the target calendar. If *target* is given as an array, the output is reindexed to it, with fill value *missing*. If *target* was a string and *missing* was None (default), invalid dates in the new calendar are dropped, but missing dates are not inserted. If *target* was a string and *missing* was given, then start, end and frequency of the new time axis are inferred and the output is reindexed to that a new array.

Notes

If one of the source or target calendars is *360_day*, *align_on* must be specified and two options are offered.

"year"

The dates are translated according to their rank in the year (dayofyear), ignoring their original month and day information, meaning that the missing/surplus days are added/removed at regular intervals.

From a *360_day* to a standard calendar, the output will be missing the following dates (day of year in parenthesis):

To a leap year:
January 31st (31), March 31st (91), June 1st (153), July 31st (213), September 31st (275) and November 30th (335).

To a non-leap year:
February 6th (36), April 19th (109), July 2nd (183), September 12th (255), November 25th (329).

From standard calendar to a '360_day', the following dates in the source array will be dropped:

From a leap year:
January 31st (31), April 1st (92), June 1st (153), August 1st (214), September 31st (275), December 1st (336)

From a non-leap year:
February 6th (37), April 20th (110), July 2nd (183), September 13th (256), November 25th (329)

This option is best used on daily and subdaily data.

"date"

The month/day information is conserved and invalid dates are dropped from the output. This means that when converting from a *360_day* to a standard calendar, all 31st (Jan, March, May, July, August, October and December) will be missing as there is no equivalent dates in the *360_day* and the 29th (on non-leap years) and 30th of February will be dropped as there are no equivalent dates in a standard calendar.

This option is best used with data on a frequency coarser than daily.

"random"

Similar to "year", each day of year of the source is mapped to another day of year of the target. However, instead of having always the same missing days according the source and target years, here 5 days are chosen randomly, one for each fifth of the year. However, February 29th is always missing when converting to a leap year, or its value is dropped when converting from a leap year. This is similar to method used in the [LOCA] dataset.

This option best used on daily data.

References

Examples

This method does not try to fill the missing dates other than with a constant value, passed with *missing*. In order to fill the missing dates with interpolation, one can simply use xarray's method:

```
>>> tas_nl = convert_calendar(tas, "noleap")  # For the example
>>> with_missing = convert_calendar(tas_nl, "standard", missing=np.NaN)
>>> out = with_missing.interpolate_na("time", method="linear")
```

Here, if Nans existed in the source data, they will be interpolated too. If that is, for some reason, not wanted, the workaround is to do:

```
>>> mask = convert_calendar(tas_nl, "standard").notnull()
>>> out2 = out.where(mask)
```

`xclim.core.calendar.date_range(`**args*, *calendar: str = 'default'*, ***kwargs*`)` → pd.DatetimeIndex | CFTimeIndex

Wrap pd.date_range (if calendar == 'default') or xr.cftime_range (otherwise).

`xclim.core.calendar.date_range_like(`*source: DataArray*, *calendar: str*`)` → DataArray

Generate a datetime array with the same frequency, start and end as another one, but in a different calendar.

Parameters

- **source** (*xr.DataArray*) – 1D datetime coordinate DataArray
- **calendar** (*str*) – New calendar name.

Raises

`ValueError` – If the source's frequency was not found.

Returns

xr.DataArray –

1D datetime coordinate with the same start, end and frequency as the source, but in the new calendar.

The start date is assumed to exist in the target calendar. If the end date doesn't exist, the code tries 1 and 2 calendar days before. Exception when the source is in 360_day and the end of the range is the 30th of a 31-days month, then the 31st is appended to the range.

`xclim.core.calendar.datetime_to_decimal_year(`*times: DataArray*, *calendar: str = ''*`)` → DataArray

Convert a datetime xr.DataArray to decimal years according to its calendar or the given one.

Decimal years are the number of years since 0001-01-01 00:00:00 AD. Ex: '2000-03-01 12:00' is 2000.1653 in a standard calendar, 2000.16301 in a "noleap" or 2000.16806 in a "360_day".

`xclim.core.calendar.days_in_year(`*year: int*, *calendar: str = 'default'*`)` → int

Return the number of days in the input year according to the input calendar.

`xclim.core.calendar.days_since_to_doy(`*da: xr.DataArray*, *start: DayOfYearStr | None = None*, *calendar: str | None = None*`)` → xr.DataArray

Reverse the conversion made by `doy_to_days_since()`.

Converts data given in days since a specific date to day-of-year.

Parameters

- **da** (*xr.DataArray*) – The result of `doy_to_days_since()`.
- **start** (*DateOfYearStr, optional*) – *da* is considered as days since that start date (in the year of the time index). If None (default), it is read from the attributes.
- **calendar** (*str, optional*) – Calendar the "days since" were computed in. If None (default), it is read from the attributes.

Returns

xr.DataArray – Same shape as *da*, values as *day of year*.

Examples

```
>>> from xarray import DataArray
>>> time = date_range("2020-07-01", "2021-07-01", freq="AS-JUL")
>>> da = DataArray(
...     [-86, 92],
...     dims=("time",),
...     coords={"time": time},
...     attrs={"units": "days since 10-02"},
... )
>>> days_since_to_doy(da).values
array([190, 2])
```

`xclim.core.calendar.doy_to_days_since(`*da: xr.DataArray, start: DayOfYearStr | None = None, calendar: str | None = None*`)` → xr.DataArray

Convert day-of-year data to days since a given date.

This is useful for computing meaningful statistics on doy data.

Parameters

- **da** (*xr.DataArray*) – Array of "day-of-year", usually int dtype, must have a *time* dimension. Sampling frequency should be finer or similar to yearly and coarser then daily.
- **start** (*date of year str, optional*) – A date in "MM-DD" format, the base day of the new array. If None (default), the *time* axis is used. Passing *start* only makes sense if *da* has a yearly sampling frequency.
- **calendar** (*str, optional*) – The calendar to use when computing the new interval. If None (default), the calendar attribute of the data or of its *time* axis is used. All time coordinates of *da* must exist in this calendar. No check is done to ensure doy values exist in this calendar.

Returns

xr.DataArray – Same shape as *da*, int dtype, day-of-year data translated to a number of days since a given date. If start is not None, there might be negative values.

Notes

The time coordinates of *da* are considered as the START of the period. For example, a doy value of 350 with a timestamp of '2020-12-31' is understood as '2021-12-16' (the 350th day of 2021). Passing *start=None*, will use the time coordinate as the base, so in this case the converted value will be 350 "days since time coordinate".

Examples

```
>>> from xarray import DataArray
>>> time = date_range("2020-07-01", "2021-07-01", freq="AS-JUL")
>>> # July 8th 2020 and Jan 2nd 2022
>>> da = DataArray([190, 2], dims=("time",), coords={"time": time})
>>> # Convert to days since Oct. 2nd, of the data's year.
>>> doy_to_days_since(da, start="10-02").values
array([-86, 92])
```

`xclim.core.calendar.ensure_cftime_array`(*time: Sequence*) → ndarray

Convert an input 1D array to a numpy array of cftime objects.

Python's datetime are converted to cftime.DatetimeGregorian ("standard" calendar).

Raises ValueError when unable to cast the input.

`xclim.core.calendar.get_calendar`(*obj: Any*, *dim: str = 'time'*) → str

Return the calendar of an object.

Parameters

- **obj** (*Any*) – An object defining some date. If *obj* is an array/dataset with a datetime coordinate, use *dim* to specify its name. Values must have either a datetime64 dtype or a cftime dtype. *obj* can also be a python datetime.datetime, a cftime object or a pandas Timestamp or an iterable of those, in which case the calendar is inferred from the first value.
- **dim** (*str*) – Name of the coordinate to check (if *obj* is a DataArray or Dataset).

Raises

`ValueError` – If no calendar could be inferred.

Returns

str – The cftime calendar name or "default" when the data is using numpy's or python's datetime types. Will always return "standard" instead of "gregorian", following CF conventions 1.9.

`xclim.core.calendar.interp_calendar`(*source: xr.DataArray | xr.Dataset*, *target: xr.DataArray*, *dim: str = 'time'*) → xr.DataArray | xr.Dataset

Interpolates a DataArray/Dataset to another calendar based on decimal year measure.

Each timestamp in source and target are first converted to their decimal year equivalent then source is interpolated on the target coordinate. The decimal year is the number of years since 0001-01-01 AD. Ex: '2000-03-01 12:00' is 2000.1653 in a standard calendar or 2000.16301 in a 'noleap' calendar.

This method should be used with daily data or coarser. Sub-daily result will have a modified day cycle.

Parameters

- **source** (*Union[xr.DataArray, xr.Dataset]*) – The source data to interpolate, must have a time coordinate of a valid dtype (np.datetime64 or cftime objects)

- **target** (*xr.DataArray*) – The target time coordinate of a valid dtype (np.datetime64 or cftime objects)
- **dim** (*str*) – The time coordinate name.

Returns
Union[xr.DataArray, xr.Dataset] – The source interpolated on the decimal years of target,

`xclim.core.calendar.parse_offset(`*freq: str*`)` → Sequence[str]

Parse an offset string.

Parse a frequency offset and, if needed, convert to cftime-compatible components.

Parameters
freq (*str*) – Frequency offset.

Returns
multiplicator (int), offset base (str), is start anchored (bool), anchor (str or None) – "[n]W" is always replaced with "[7n]D", as xarray doesn't support "W" for cftime indexes. "Y" is always replaced with "A".

`xclim.core.calendar.percentile_doy(`*arr: xr.DataArray, window: int = 5, per: float | Sequence[float] = 10.0, alpha: float = 0.3333333333333333, beta: float = 0.3333333333333333, copy: bool = True*`)` → *PercentileDataArray*

Percentile value for each day of the year.

Return the climatological percentile over a moving window around each day of the year. Different quantile estimators can be used by specifying *alpha* and *beta* according to specifications given by [HyndmanFan]. The default definition corresponds to method 8, which meets multiple desirable statistical properties for sample quantiles. Note that *numpy.percentile* corresponds to method 7, with alpha and beta set to 1.

Parameters

- **arr** (*xr.DataArray*) – Input data, a daily frequency (or coarser) is required.
- **window** (*int*) – Number of time-steps around each day of the year to include in the calculation.
- **per** (*float or sequence of floats*) – Percentile(s) between [0, 100]
- **alpha** (*float*) – Plotting position parameter.
- **beta** (*float*) – Plotting position parameter.
- **copy** (*bool*) – If True (default) the input array will be deep copied. It's a necessary step to keep the data integrity but it can be costly. If False, no copy is made of the input array. It will be mutated and rendered unusable but performances may significantly improve. Put this flag to False only if you understand the consequences.

Returns
xr.DataArray – The percentiles indexed by the day of the year. For calendars with 366 days, percentiles of doys 1-365 are interpolated to the 1-366 range.

References

`xclim.core.calendar.resample_doy(`*doy: xr.DataArray*, *arr: xr.DataArray | xr.Dataset*`)` → xr.DataArray

Create a temporal DataArray where each day takes the value defined by the day-of-year.

Parameters

- **doy** (*xr.DataArray*) – Array with *dayofyear* coordinate.
- **arr** (*xr.DataArray or xr.Dataset*) – Array with *time* coordinate.

Returns

xr.DataArray – An array with the same dimensions as *doy*, except for *dayofyear*, which is replaced by the *time* dimension of *arr*. Values are filled according to the day of year value in *doy*.

`xclim.core.calendar.select_time(`*da: xr.DataArray | xr.Dataset*, *drop: bool = False*, *season: str | Sequence[str] = None*, *month: int | Sequence[int] = None*, *doy_bounds: tuple[int, int] = None*, *date_bounds: tuple[str, str] = None*`)` → xr.DataArray | xr.Dataset

Select entries according to a time period.

This conveniently improves xarray's `xarray.DataArray.where()` and `xarray.DataArray.sel()` with fancier ways of indexing over time elements. In addition to the data *da* and argument *drop*, only one of *season*, *month*, *doy_bounds* or *date_bounds* may be passed.

Parameters

- **da** (*xr.DataArray or xr.Dataset*) – Input data.
- **drop** (*boolean*) – Whether to drop elements outside the period of interest or to simply mask them (default).
- **season** (*string or sequence of strings*) – One or more of 'DJF', 'MAM', 'JJA' and 'SON'.
- **month** (*integer or sequence of integers*) – Sequence of month numbers (January = 1 ... December = 12)
- **doy_bounds** (*2-tuple of integers*) – The bounds as (start, end) of the period of interest expressed in day-of-year, integers going from 1 (January 1st) to 365 or 366 (December 31st). If calendar awareness is needed, consider using `date_bounds` instead. Bounds are inclusive.
- **date_bounds** (*2-tuple of strings*) – The bounds as (start, end) of the period of interest expressed as dates in the month-day (%m-%d) format. Bounds are inclusive.

Returns

xr.DataArray or xr.Dataset – Selected input values. If `drop=False`, this has the same length as `da` (along dimension 'time'), but with masked (NaN) values outside the period of interest.

Examples

Keep only the values of fall and spring.

```
>>> ds = open_dataset("ERA5/daily_surface_cancities_1990-1993.nc")
>>> ds.time.size
1461
>>> out = select_time(ds, drop=True, season=["MAM", "SON"])
>>> out.time.size
732
```

Or all values between two dates (included).

```
>>> out = select_time(ds, drop=True, date_bounds=("02-29", "03-02"))
>>> out.time.values
array(['1990-03-01T00:00:00.000000000', '1990-03-02T00:00:00.000000000',
       '1991-03-01T00:00:00.000000000', '1991-03-02T00:00:00.000000000',
       '1992-02-29T00:00:00.000000000', '1992-03-01T00:00:00.000000000',
       '1992-03-02T00:00:00.000000000', '1993-03-01T00:00:00.000000000',
       '1993-03-02T00:00:00.000000000'], dtype='datetime64[ns]')
```

`xclim.core.calendar.time_bnds`(*group*, *freq: str*) → Sequence[tuple[cftime.datetime, cftime.datetime]]

Find the time bounds for a pseudo-period index.

As we are using datetime indices to stand in for period indices, assumptions regarding the period are made based on the given freq. IMPORTANT NOTE: this function cannot be used on greater-than-day freq that start at the beginning of a month, e.g. ‘MS’, ‘QS’, ‘AS’ – this mirrors pandas behavior.

Parameters

- **group** (*CFTimeIndex or DataArrayResample*) – Object which contains CFTimeIndex as a proxy representation for CFPeriodIndex
- **freq** (*str*) – String specifying the frequency/offset such as ‘MS’, ‘2D’, or ‘3T’

Returns

Sequence[(cftime.datetime, cftime.datetime)] – The start and end times of the period inferred from datetime and freq.

Examples

```
>>> from xarray import cftime_range
>>> from xclim.core.calendar import time_bnds
>>> index = cftime_range(
...     start="2000-01-01", periods=3, freq="2QS", calendar="360_day"
... )
>>> out = time_bnds(index, "2Q")
>>> for bnds in out:
...     print(
...         bnds[0].strftime("%Y-%m-%dT%H:%M:%S"),
...         " -",
...         bnds[1].strftime("%Y-%m-%dT%H:%M:%S"),
...     )
...
2000-01-01T00:00:00  - 2000-03-30T23:59:59
```

(continues on next page)

(continued from previous page)

```
2000-07-01T00:00:00  - 2000-09-30T23:59:59
2001-01-01T00:00:00  - 2001-03-30T23:59:59
```

`xclim.core.calendar.within_bnds_doy(`*arr: DataArray*, ***, *low: DataArray*, *high: DataArray*`)` → DataArray

Return whether or not array values are within bounds for each day of the year.

Parameters

- **arr** (*xarray.DataArray*) – Input array.
- **low** (*xarray.DataArray*) – Low bound with dayofyear coordinate.
- **high** (*xarray.DataArray*) – High bound with dayofyear coordinate.

Returns

xarray.DataArray

xclim.core.cfchecks module

CF-Convention checking

Utilities designed to verify the compliance of metadata with the CF-Convention.

`xclim.core.cfchecks._check_cell_methods(`*data_cell_methods: str*, *expected_method: str*`)` → None

`xclim.core.cfchecks.cfcheck_from_name(`*varname*, *vardata*`)`

Perform cfchecks on a DataArray using specifications from xclim's default variables.

`xclim.core.cfchecks.check_valid(`*var*, *key: str*, *expected: str | Sequence[str]*`)`

Check that a variable's attribute has one of the expected values. Raise a ValidationError otherwise.

xclim.core.datachecks module

Data checks

Utilities designed to check the validity of data inputs.

`xclim.core.datachecks.check_daily(`*var: DataArray*`)`

Raise an error if not series has a frequency other that daily, or is not monotonically increasing.

Note that this does not check for gaps in the series.

`xclim.core.datachecks.check_freq(`*var: xr.DataArray*, *freq: str | Sequence[str]*, *strict: bool = True*`)`

Raise an error if not series has not the expected temporal frequency or is not monotonically increasing.

Parameters

- **var** (*xr.DataArray*) – Input array.
- **freq** (*str or sequence of str*) – The expected temporal frequencies, using Pandas frequency terminology ({'A', 'M', 'D', 'H', 'T', 'S', 'L', 'U'} and multiples thereof). To test strictly for 'W', pass '7D' with *strict=True*. This ignores the start flag and the anchor (ex: 'AS-JUL' will validate against 'Y').

- **strict** (*bool*) – Whether multiples of the frequencies are considered invalid or not. With *strict* set to False, a '3H' series will not raise an error if freq is set to 'H'.

xclim.core.dataflags module

Data flags

Pseudo-indicators designed to analyse supplied variables for suspicious/erroneous indicator values.

`exception xclim.core.dataflags.DataQualityException(`*flag_array: Dataset, message='Data quality flags indicate suspicious values. Flags raised are:\n - '*`)`

Bases: `Exception`

Raised when any data evaluation checks are flagged as True.

Variables

- `flag_array` (*`xarray.Dataset`*) – Xarray.Dataset of Data Flags.
- `message` (*`str`*) – Message prepended to the error messages.

`xclim.core.dataflags.data_flags(`*da: xarray.DataArray, ds: xarray.Dataset | None = None, flags: dict | None = None, dims: None | str | Sequence[str] = 'all', freq: str | None = None, raise_flags: bool = False*`)` → xarray.Dataset

Evaluate the supplied DataArray for a set of data flag checks.

Test triggers depend on variable name and availability of extra variables within Dataset for comparison. If called with *raise_flags=True*, will raise a DataQualityException with comments for each failed quality check.

Parameters

- **da** (*xarray.DataArray*) – The variable to check. Must have a name that is a valid CMIP6 variable name and appears in `xclim.core.utils.VARIABLES`.
- **ds** (*xarray.Dataset, optional*) – An optional dataset with extra variables needed by some checks.
- **flags** (*dict, optional*) – A dictionary where the keys are the name of the flags to check and the values are parameter dictionaries. The value can be None if there are no parameters to pass (i.e. default will be used). The default, None, means that the data flags list will be taken from `xclim.core.utils.VARIABLES`.
- **dims** (*{"all", None} or str or a sequence of strings*) – Dimenions upon which aggregation should be performed. Default: "all".
- **freq** (*str, optional*) – Resampling frequency to have data_flags aggregated over periods. Defaults to None, which means the "time" axis is treated as any other dimension (see *dims*).
- **raise_flags** (*bool*) – Raise exception if any of the quality assessment flags are raised. Default: False.

Returns

xarray.Dataset

Examples

To evaluate all applicable data flags for a given variable:

```
>>> from xclim.core.dataflags import data_flags
>>> ds = xr.open_dataset(path_to_pr_file)
>>> flagged = data_flags(ds.pr, ds)
```

The next example evaluates only one data flag, passing specific parameters. It also aggregates the flags yearly over the "time" dimension only, such that a True means there is a bad data point for that year at that location.

```
>>> flagged = data_flags(
...     ds.pr,
...     ds,
...     flags={"very_large_precipitation_events": {"thresh": "250 mm d-1"}},
...     dims=None,
...     freq="YS",
... )
```

`xclim.core.dataflags.ecad_compliant(`*ds: xarray.Dataset*, *dims: None | str | Sequence[str] = 'all'*, *raise_flags: bool = False*, *append: bool = True*`)` → xarray.DataArray | xarray.Dataset | None

Run ECAD compliance tests.

Assert file adheres to ECAD-based quality assurance checks.

Parameters

- **ds** (*xarray.Dataset*) – Dataset containing variables to be examined.
- **dims** (*{"all", None} or str or a sequence of strings*) – Dimensions upon which aggregation should be performed. Default: "all".
- **raise_flags** (*bool*) – Raise exception if any of the quality assessment flags are raised, otherwise returns None. Default: False.
- **append** (*bool*) – If *True*, returns the Dataset with the *ecad_qc_flag* array appended to data_vars. If *False*, returns the DataArray of the *ecad_qc_flag* variable.

Returns

Union[xarray.DataArray, xarray.Dataset]

`xclim.core.dataflags.negative_accumulation_values(`*da: DataArray*`)` → DataArray

Check if variable values are negative for any given day.

Parameters

da (*xarray.DataArray*)

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import negative_accumulation_values
>>> ds = xr.open_dataset(path_to_pr_file)
>>> flagged = negative_accumulation_values(ds.pr)
```

`xclim.core.dataflags.outside_n_standard_deviations_of_climatology`(*da: DataArray*, ***, *n: int*, *window: int = 5*) → DataArray

Check if any daily value is outside n standard deviations from the day of year mean.

Parameters

- **da** (*xarray.DataArray*) – The DataArray being examined.
- **n** (*int*) – Number of standard deviations.
- **window** (*int*) – Moving window used to determining climatological mean. Default: 5.

Returns

xarray.DataArray, [bool]

Notes

A moving window of 5 days is suggested for tas data flag calculations according to ICCLIM data quality standards.

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import outside_n_standard_deviations_of_climatology
>>> ds = xr.open_dataset(path_to_tas_file)
>>> std_devs = 5
>>> average_over = 5
>>> flagged = outside_n_standard_deviations_of_climatology(
...     ds.tas, n=std_devs, window=average_over
... )
```

`xclim.core.dataflags.percentage_values_outside_of_bounds`(*da: DataArray*) → DataArray

Check if variable values fall below 0% or rise above 100% for any given day.

Parameters

da (*xarray.DataArray*)

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array: >>> from xclim.core.dataflags import percentage_values_outside_of_bounds >>> ds = xr.open_dataset(path_to_huss_file) # doctest: +SKIP >>> flagged = percentage_values_outside_of_bounds(ds.huss) # doctest: +SKIP

`xclim.core.dataflags.register_methods(`*func*`)`

Summarize all methods used in dataflags checks.

`xclim.core.dataflags.tas_below_tasmin(`*tas: DataArray*, *tasmin: DataArray*`)` → DataArray

Check if tas values are below tasmin values for any given day.

Parameters

- **tas** (*xarray.DataArray*)
- **tasmin** (*xarray.DataArray*)

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import tas_below_tasmin
>>> ds = xr.open_dataset(path_to_tas_file)
>>> flagged = tas_below_tasmin(ds.tas, ds.tasmin)
```

`xclim.core.dataflags.tas_exceeds_tasmax(`*tas: DataArray*, *tasmax: DataArray*`)` → DataArray

Check if tas values tasmax values for any given day.

Parameters

- **tas** (*xarray.DataArray*)
- **tasmax** (*xarray.DataArray*)

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import tas_exceeds_tasmax
>>> ds = xr.open_dataset(path_to_tas_file)
>>> flagged = tas_exceeds_tasmax(ds.tas, ds.tasmax)
```

`xclim.core.dataflags.tasmax_below_tasmin(`*tasmax: DataArray*, *tasmin: DataArray*`)` → DataArray

Check if tasmax values are below tasmin values for any given day.

Parameters

- **tasmax** (*xarray.DataArray*)
- **tasmin** (*xarray.DataArray*)

Returns
xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import tasmax_below_tasmin
>>> ds = xr.open_dataset(path_to_tas_file)
>>> flagged = tasmax_below_tasmin(ds.tasmax, ds.tasmin)
```

`xclim.core.dataflags.temperature_extremely_high(`*da: DataArray*, **, *thresh: str = '60 degC'*`)` → DataArray

Check if temperatures values exceed 60 degrees Celsius for any given day.

Parameters

- **da** (*xarray.DataArray*)
- **thresh** (*str*)

Returns
xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import temperature_extremely_high
>>> ds = xr.open_dataset(path_to_tas_file)
>>> temperature = "60 degC"
>>> flagged = temperature_extremely_high(ds.tas, thresh=temperature)
```

`xclim.core.dataflags.temperature_extremely_low(`*da: DataArray*, **, *thresh: str = '-90 degC'*`)` → DataArray

Check if temperatures values are below -90 degrees Celsius for any given day.

Parameters

- **da** (*xarray.DataArray*)
- **thresh** (*str*)

Returns
xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import temperature_extremely_low
>>> ds = xr.open_dataset(path_to_tas_file)
>>> temperature = "-90 degC"
>>> flagged = temperature_extremely_low(ds.tas, thresh=temperature)
```

`xclim.core.dataflags.values_op_thresh_repeating_for_n_or_more_days(`*da: DataArray*, *, *n: int*, *thresh: str*, *op: str = 'eq'*`)` → DataArray

Check if array values repeat at a given threshold for 'n' or more days.

Parameters

- **da** (*xarray.DataArray*) – The DataArray being examined.
- **n** (*int*) – Number of days needed to trigger flag.
- **thresh** (*str*) – Repeating values to search for that will trigger flag.
- **op** (*{"eq", "gt", "lt", "gteq", "lteq"}*) – Operator used for comparison with thresh.

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import values_op_thresh_repeating_for_n_or_more_days
>>> ds = xr.open_dataset(path_to_pr_file)
>>> units = "5 mm d-1"
>>> days = 5
>>> comparison = "eq"
>>> flagged = values_op_thresh_repeating_for_n_or_more_days(
...     ds.pr, n=days, thresh=units, op=comparison
... )
```

`xclim.core.dataflags.values_repeating_for_n_or_more_days(`*da: DataArray*, *, *n: int*`)` → DataArray

Check if exact values are found to be repeating for at least 5 or more days.

Parameters

- **da** (*xarray.DataArray*) – The DataArray being examined.
- **n** (*int*) – Number of days to trigger flag.

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import values_repeating_for_n_or_more_days
>>> ds = xr.open_dataset(path_to_pr_file)
>>> flagged = values_repeating_for_n_or_more_days(ds.pr, n=5)
```

`xclim.core.dataflags.very_large_precipitation_events`(*da: DataArray*, ***, *thresh='300 mm d-1'*) → DataArray

Check if precipitation values exceed 300 mm/day for any given day.

Parameters

- **da** (*xarray.DataArray*) – The DataArray being examined.
- **thresh** (*str*) – Threshold to search array for that will trigger flag if any day exceeds value.

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array:

```
>>> from xclim.core.dataflags import very_large_precipitation_events
>>> ds = xr.open_dataset(path_to_pr_file)
>>> rate = "300 mm d-1"
>>> flagged = very_large_precipitation_events(ds.pr, thresh=rate)
```

`xclim.core.dataflags.wind_values_outside_of_bounds`(*da: DataArray*, ***, *lower: str = '0 m s-1'*, *upper: str = '46 m s-1'*) → DataArray

Check if variable values fall below 0% or rise above 100% for any given day.

Parameters

- **da** (*xarray.DataArray*) – The DataArray being examined.
- **lower** (*str*) – The lower limit for wind speed.
- **upper** (*str*) – The upper limit for wind speed.

Returns

xarray.DataArray, [bool]

Examples

To gain access to the flag_array: >>> from xclim.core.dataflags import wind_values_outside_of_bounds >>> ds = xr.open_dataset(path_to_tas_file) >>> ceiling, floor = “46 m s-1”, “0 m s-1” >>> flagged = wind_values_outside_of_bounds(ds.wsgsmax, upper=ceiling, lower=floor)

xclim.core.formatting module

Formatting utilities for indicators

`class xclim.core.formatting.AttrFormatter(`*mapping: Mapping[str, Sequence[str]], modifiers: Sequence[str]*`)`

Bases: `Formatter`

A formatter for frequently used attribute values.

See the doc of format_field() for more details.

`_match_value(`*value*`)`

`format(`*format_string: str, /, *args: Any, **kwargs: dict*`)` → str

Format a string.

Parameters

- **format_string** (*str*)
- **args**
- **kwargs**

Returns
str

`format_field(`*value, format_spec*`)`

Format a value given a formatting spec.

If *format_spec* is in this Formatter's modifiers, the corresponding variation of value is given. If *format_spec* is 'r' (raw), the value is returned unmodified. If *format_spec* is not specified but *value* is in the mapping, the first variation is returned.

Examples

Let's say the string "The dog is {adj1}, the goose is {adj2}" is to be translated to french and that we know that possible values of *adj* are *nice* and *evil*. In french, the genre of the noun changes the adjective (cat = chat is masculine, and goose = oie is feminine) so we initialize the formatter as:

```
>>> fmt = AttrFormatter(
...     {
...         "nice": ["beau", "belle"],
...         "evil": ["méchant", "méchante"],
...         "smart": ["intelligent", "intelligente"],
...     },
...     ["m", "f"],
... )
>>> fmt.format(
...     "Le chien est {adj1:m}, l'oie est {adj2:f}, le gecko est {adj3:r}",
...     adj1="nice",
...     adj2="evil",
...     adj3="smart",
... )
"Le chien est beau, l'oie est méchante, le gecko est smart"
```

The base values may be given using unix shell-like patterns:

```
>>> fmt = AttrFormatter(
...     {"AS-*": ["annuel", "annuelle"], "MS": ["mensuel", "mensuelle"]},
...     ["m", "f"],
... )
>>> fmt.format(
...     "La moyenne {freq:f} est faite sur un échantillon {src_timestep:m}",
...     freq="AS-JUL",
...     src_timestep="MS",
... )
'La moyenne annuelle est faite sur un échantillon mensuel'
```

`xclim.core.formatting._gen_parameters_section`(*parameters: Mapping*, *allowed_periods: list[str] = None*)

Generate the "parameters" section of the indicator docstring.

Parameters

- **parameters** (*mapping*) – Parameters dictionary (*Ind.parameters*).
- **allowed_periods** (*List[str], optional*) – Restrict parameters to specific periods. Default: None.

`xclim.core.formatting._gen_returns_section`(*cf_attrs: Sequence[dict[str, Any]]*)

Generate the "Returns" section of an indicator's docstring.

Parameters

cf_attrs (*Sequence[Dict[str, Any]]*) – The list of attributes, usually Indicator.cf_attrs.

`xclim.core.formatting._parse_parameters`(*section*)

Parse the 'parameters' section of a docstring into a dictionary mapping the parameter name to its description and, potentially, to its set of choices.

The type annotation are not parsed, except for fixed sets of values (listed as "{'a', 'b', 'c'}"). The annotation parsing only accepts strings, numbers, *None* and *nan* (to represent *numpy.nan*).

`xclim.core.formatting._parse_returns`(*section*)

Parse the returns section of a docstring into a dictionary mapping the parameter name to its description.

`xclim.core.formatting.gen_call_string`(*funcname: str*, **args*, ***kwargs*)

Generate a signature string for use in the history attribute.

DataArrays and Dataset are replaced with their name, while Nones, floats, ints and strings are printed directly. All other objects have their type printed between < >.

Arguments given through positional arguments are printed positionnally and those given through keywords are printed prefixed by their name.

Parameters

- **funcname** (*str*) – Name of the function
- **args, kwargs** – Arguments given to the function.

Example

```
>>> A = xr.DataArray([1], dims=("x",), name="A")
>>> gen_call_string("func", A, b=2.0, c="3", d=[4, 5, 6])
"func(A, b=2.0, c='3', d=<list>)"
```

`xclim.core.formatting.generate_indicator_docstring(`*ind*`)`

Generate an indicator's docstring from keywords.

Parameters

ind (*Indicator instance*)

`xclim.core.formatting.get_percentile_metadata(`*data: xr.DataArray*, *prefix: str*`)` → dict[str, str]

Get the metadata related to percentiles from the given DataArray as a dictionary.

Parameters

- **data** (*xr.DataArray*) – Must be compatible with PercentileDataArray, this means the necessary metadata must be available in its attributes and coordinates.
- **prefix** (*str*) – The prefix to be used in the metadata key. Usually this takes the form of "tasmin_per" or equivalent.

Returns

dict – A mapping of the configuration used to compute these percentiles.

`xclim.core.formatting.merge_attributes(`*attribute: str*, **inputs_list: xr.DataArray | xr.Dataset*, *new_line: str = '\n'*, *missing_str: str | None = None*, ***inputs_kws: xr.DataArray | xr.Dataset*`)`

Merge attributes from several DataArrays or Datasets.

If more than one input is given, its name (if available) is prepended as: "<input name> : <input attribute>".

Parameters

- **attribute** (*str*) – The attribute to merge.
- **inputs_list** (*Union[xr.DataArray, xr.Dataset]*) – The datasets or variables that were used to produce the new object. Inputs given that way will be prefixed by their *name* attribute if available.
- **new_line** (*str*) – The character to put between each instance of the attributes. Usually, in CF-conventions, the history attributes uses '\n' while cell_methods uses ' '.
- **missing_str** (*str*) – A string that is printed if an input doesn't have the attribute. Defaults to None, in which case the input is simply skipped.
- **inputs_kws** (*Union[xr.DataArray, xr.Dataset]*) – Mapping from names to the datasets or variables that were used to produce the new object. Inputs given that way will be prefixes by the passed name.

Returns

str – The new attribute made from the combination of the ones from all the inputs.

`xclim.core.formatting.parse_doc(`*doc: str*`)` → dict[str, str]

Crude regex parsing reading an indice docstring and extracting information needed in indicator construction.

The appropriate docstring syntax is detailed in *Defining new indices*.

Parameters
doc (*str*) – The docstring of an indice function.

Returns
dict – A dictionary with all parsed sections.

`xclim.core.formatting.prefix_attrs(`*source: dict*, *keys: Sequence*, *prefix: str*`)`

Rename some keys of a dictionary by adding a prefix.

Parameters

- **source** (*dict*) – Source dictionary, for example data attributes.
- **keys** (*sequence*) – Names of keys to prefix.
- **prefix** (*str*) – Prefix to prepend to keys.

Returns
dict – Dictionary of attributes with some keys prefixed.

`xclim.core.formatting.unprefix_attrs(`*source: dict*, *keys: Sequence*, *prefix: str*`)`

Remove prefix from keys in a dictionary.

Parameters

- **source** (*dict*) – Source dictionary, for example data attributes.
- **keys** (*sequence*) – Names of original keys for which prefix should be removed.
- **prefix** (*str*) – Prefix to remove from keys.

Returns
dict – Dictionary of attributes whose keys were prefixed, with prefix removed.

`xclim.core.formatting.update_history(`*hist_str: str*, **inputs_list: Sequence[xr.DataArray | xr.Dataset]*, *new_name: str | None = None*, ***inputs_kws: Mapping[str, xr.DataArray | xr.Dataset]*`)`

Return a history string with the timestamped message and the combination of the history of all inputs.

The new history entry is formatted as "[<timestamp>] <new_name>: <hist_str> - xclim version: <xclim.__version__>."

Parameters

- **hist_str** (*str*) – The string describing what has been done on the data.
- **new_name** (*Optional[str]*) – The name of the newly created variable or dataset to prefix hist_msg.
- **inputs_list** (*Sequence[Union[xr.DataArray, xr.Dataset]]*) – The datasets or variables that were used to produce the new object. Inputs given that way will be prefixed by their "name" attribute if available.
- **inputs_kws** (*Mapping[str, Union[xr.DataArray, xr.Dataset]]*) – Mapping from names to the datasets or variables that were used to produce the new object. Inputs given that way will be prefixes by the passed name.

Returns
str – The combine history of all inputs starting with *hist_str*.

See also:

`merge_attributes`

`xclim.core.formatting.update_xclim_history(`*func*`)`

Decorator that auto-generates and fills the history attribute.

The history is generated from the signature of the function and added to the first output. Because of a limitation of the *boltons* wrapper, all arguments passed to the wrapped function will be printed as keyword arguments.

xclim.core.indicator module

Indicators utilities

The *Indicator* class wraps indices computations with pre- and post-processing functionality. Prior to computations, the class runs data and metadata health checks. After computations, the class masks values that should be considered missing and adds metadata attributes to the object.

There are many ways to construct indicators. A good place to start is *this notebook*.

Dictionary and YAML parser

To construct indicators dynamically, xclim can also use dictionaries and parse them from YAML files. This is especially useful for generating whole indicator "submodules" from files. This functionality is inspired by the work of clix-meta.

YAML file structure

Indicator-defining yaml files are structured in the following way. Most entries of the *indicators* section are mirroring attributes of the `Indicator`, please refer to its documentation for more details on each.

```
module: <module name>  # Defaults to the file name
realm: <realm>  # If given here, applies to all indicators that do not already provide␣
↪it.
keywords: <keywords> # Merged with indicator-specific keywords (joined with a space)
references: <references> # Merged with indicator-specific references (joined with a new␣
↪line)
base: <base indicator class>  # Defaults to "Daily" and applies to all indicators that␣
↪do not give it.
doc: <module docstring>  # Defaults to a minimal header, only valid if the module doesn
↪'t already exists.
indicators:
  <identifier>:
    # From which Indicator to inherit
    base: <base indicator class>  # Defaults to module-wide base class
                                  # If the name startswith a '.', the base class is␣
↪taken from the current module (thus an indicator declared _above_)
                                  # Available classes are listed in `xclim.core.
↪indicator.registry` and `xclim.core.indicator.base_registry`.

    # General metadata, usually parsed from the `compute`'s docstring when possible.
    realm: <realm>  # defaults to module-wide realm. One of "atmos", "land", "seaIce",
↪"ocean".
    title: <title>
```

(continues on next page)

(continued from previous page)

```
    abstract: <abstract>
    keywords: <keywords>  # Space-separated, merged to module-wide keywords.
    references: <references>  # newline-seperated, merged to module-wide references.
    notes: <notes>

    # Other options
    missing: <missing method name>
    missing_options:
        # missing options mapping
    allowed_periods: [<list>, <of>, <allowed>, <periods>]

    # Compute function
    compute: <function name>  # Referring to a function in the passed indices module,␣
↪xclim.indices.generic or xclim.indices
    input:  # When "compute" is a generic function this is a mapping from argument
            # name to what CMIP6/xclim variable is expected. This will allow for
            # declaring expected input units and have a CF metadata check on the inputs.
            # Can also be used to modify the expected variable, as long as it has
            # the same units. Ex: tas instead of tasmin.
      <var name in compute> : <variable official name>
      ...
    parameters:
     <param name>: <param data>  # Simplest case, to inject parameters in the compute␣
↪function.
     <param name>:  # To change parameters metadata or to declare units when "compute"␣
↪is a generic function.
        units: <param units>  # Only valid if "compute" points to a generic function
        default : <param default>
        description: <param description>
    ...
  ...  # and so on.
```

All fields are optional. Other fields found in the yaml file will trigger errors in xclim. In the following, the section under *<identifier>* is refered to as *data*. When creating indicators from a dictionary, with *Indicator.from_dict()*, the input dict must follow the same structure of *data*.

The resulting yaml file can be validated using the provided schema (in xclim/data/schema.yml) and the yamale tool. See the "Extending xclim" notebook for more info.

Inputs

As xclim has strict definitions of possible input variables (see `xclim.core.utils.variables`), the mapping of *data.input* simply links an argument name from the function given in "compute" to one of those official variables.

class xclim.core.indicator.Daily(***kwds*)

Bases: *ResamplingIndicator*

Class for daily inputs and resampling computes.

src_freq = 'D'

`class xclim.core.indicator.Hourly(**kwds)`

Bases: *ResamplingIndicator*

Class for hourly inputs and resampling computes.

`src_freq = 'H'`

`class xclim.core.indicator.Indicator(**kwds)`

Bases: *IndicatorRegistrar*

Climate indicator base class.

Climate indicator object that, when called, computes an indicator and assigns its output a number of CF-compliant attributes. Some of these attributes can be *templated*, allowing metadata to reflect the value of call arguments.

Instantiating a new indicator returns an instance but also creates and registers a custom subclass in `xclim.core.indicator.registry`.

Attributes in *Indicator.cf_attrs* will be formatted and added to the output variable(s). This attribute is a list of dictionaries. For convenience and retro-compatibility, standard CF attributes (names listed in *xclim.core.indicator.Indicator._cf_names*) can be passed as strings or list of strings directly to the indicator constructor.

A lot of the Indicator's metadata is parsed from the underlying *compute* function's docstring and signature. Input variables and parameters are listed in *xclim.core.indicator.Indicator.parameters*, while parameters that will be injected in the compute function are in *xclim.core.indicator.Indicator.injected_parameters*. Both are simply views of *xclim.core.indicator.Indicator._all_parameters*.

Compared to their base *compute* function, indicators add the possibility of using dataset as input, with the injected argument *ds* in the call signature. All arguments that were indicated by the compute function to be variables (DataArrays) through annotations will be promoted to also accept strings that correspond to variable names in the *ds* dataset.

Parameters

- **identifier** (*str*) – Unique ID for class registry, should be a valid slug.
- **realm** (*{'atmos', 'seaIce', 'land', 'ocean'}*) – General domain of validity of the indicator. Indicators created outside xclim.indicators must set this attribute.
- **compute** (*func*) – The function computing the indicators. It should return one or more DataArray.
- **cf_attrs** (*list of dicts*) – Attributes to be formatted and added to the computation's output. See *xclim.core.indicator.Indicator.cf_attrs*.
- **title** (*str*) – A succinct description of what is in the computed outputs. Parsed from *compute* docstring if None (first paragraph).
- **abstract** (*str*) – A long description of what is in the computed outputs. Parsed from *compute* docstring if None (second paragraph).
- **keywords** (*str*) – Comma separated list of keywords. Parsed from *compute* docstring if None (from a "Keywords" section).
- **references** (*str*) – Published or web-based references that describe the data or methods used to produce it. Parsed from *compute* docstring if None (from the "References" section).
- **notes** (*str*) – Notes regarding computing function, for example the mathematical formulation. Parsed from *compute* docstring if None (form the "Notes" section).

- **src_freq** (*str, sequence of strings, optional*) – The expected frequency of the input data. Can be a list for multiple frequencies, or None if irrelevant.
- **context** (*str*) – The *pint* unit context, for example use 'hydro' to allow conversion from kg m-2 s-1 to mm/day.

Notes

All subclasses created are available in the *registry* attribute and can be used to define custom subclasses or parse all available instances.

`_all_parameters: Mapping[str, Parameter] = {}`

A dictionary mapping metadata about the input parameters to the indicator.

Keys are the arguments of the "compute" function. All parameters are listed, even those "injected", absent from the indicator's call signature. All are instances of `xclim.core.indicator.Parameter`.

`_assign_named_args(`*ba*`)`

Assign inputs passed as strings from ds.

`_bind_call(`*func*, ***das*`)`

Call function using *__call__ DataArray* arguments.

This will try to bind keyword arguments to *func* arguments. If this fails, *func* is called with positional arguments only.

Notes

This method is used to support two main use cases.

In use case #1, we have two compute functions with arguments in a different order:
func1(tasmin, tasmax) and *func2(tasmax, tasmin)*

In use case #2, we have two compute functions with arguments that have different names:
generic_func(da) and *custom_func(tas)*

For each case, we want to define a single *cfcheck* and *datacheck* methods that will work with both compute functions.

Passing a dictionary of arguments will solve #1, but not #2.

`_cf_names = ['var_name', 'standard_name', 'long_name', 'units', 'cell_methods', 'description', 'comment']`

`static _check_identifier(`*identifier: str*`)` → None

Verify that the identifier is a proper slug.

`classmethod _ensure_correct_parameters(`*parameters*`)`

Ensure the parameters are correctly set and ordered.

Sets the correct variable default to be sure.

`classmethod _format(`*attrs: dict*, *args: ~typing.Optional[dict] = None*, *formatter: ~xclim.core.formatting.AttrFormatter = <xclim.core.formatting.AttrFormatter object>*`)`

Format attributes including {} tags with arguments.

Parameters

- **attrs** (*dict*) – Attributes containing tags to replace with arguments' values.
- **args** (*dict, optional*) – Function call arguments. If not given, the default arguments will be used when formatting the attributes.
- **formatter** (*AttrFormatter*) – Plaintext mappings for indicator attributes.

`_funcs = ['compute']`

`_gen_signature()`

Generate the correct signature.

`classmethod _get_translated_metadata(`*locale*, *var_id=None*, *names=None*, *append_locale_name=True*`)`

Get raw translated metadata for the curent indicator and a given locale.

All available translated metadata from the current indicator and those it is based on are merged, with highest priority to the current one.

`_history_string(`***kwargs*`)`

`classmethod _injected_parameters()`

Create a list of tuples for arguments to inject, (name, Parameter).

`static _parse_indice(`*compute*, *passed_parameters*`)`

Parse the compute function.

- Metadata is extracted from the docstring
- Parameters are parsed from the docstring (description, choices), decorator (units), signature (kind, default)

'passed_parameters' is only needed when compute is a generic function (not decorated by *declare_units*) and it takes a string parameter. In that case we need to check if that parameter has units (which have been passed explicitly).

`classmethod _parse_output_attrs(`*kwds: dict[str, Any]*, *identifier: str*`)` → list[dict[str, str | Callable]]

CF-compliant metadata attributes for all output variables.

`classmethod _parse_var_mapping(`*variable_mapping*, *parameters*, *kwds*`)`

Parse the variable mapping passed in *input* and update *parameters* in-place.

`_parse_variables_from_call(`*args*, *kwds*`)`

Extract variable and optional variables from call arguments.

`_postprocess(`*outs*, *das*, *params*`)`

Actions to done after computing.

`_preprocess_and_checks(`*das*, *params*`)`

Actions to be done after parsing the arguments and before computing.

`_text_fields = ['long_name', 'description', 'comment']`

`_update_attrs(`*args*, *das*, *attrs*, *var_id=None*, *names=None*`)`

Format attributes with the run-time values of *compute* call parameters.

Cell methods and history attributes are updated, adding to existing values. The language of the string is taken from the *OPTIONS* configuration dictionary.

Parameters

- **args** (*Mapping[str, Any]*) – Keyword arguments of the *compute* call.
- **das** (*Mapping[str, DataArray]*) – Input arrays.
- **attrs** (*Mapping[str, str]*) – The attributes to format and update.
- **var_id** (*str*) – The identifier to use when requesting the attributes translations. Defaults to the class name (for the translations) or the *identifier* field of the class (for the history attribute). If given, the identifier will be converted to uppercase to get the translation attributes. This is meant for multi-outputs indicators.
- **names** (*Sequence[str]*) – List of attribute names for which to get a translation.

Returns

dict – Attributes with {} expressions replaced by call argument values. With updated *cell_methods* and *history*. *cell_methods* is not added if *names* is given and those not contain *cell_methods*.

`classmethod _update_parameters(`*parameters*, *passed*`)`

Update parameters with the ones passed.

`_variable_mapping = {}`

`abstract = ''`

`cf_attrs: Sequence[Mapping[str, Any]] = None`

A list of metadata information for each output of the indicator.

It minimally contains a “var_name” entry, and may contain : “standard_name”, “long_name”, “units”, “cell_methods”, “description” and “comment” on official xclim indicators. Other fields could also be present if the indicator was created from outside xclim.

var_name:
Output variable(s) name(s).

standard_name:
Variable name, must be in the CF standard names table (this is not checked).

long_name:
Descriptive variable name. Parsed from *compute* docstring if not given. (first line after the output dtype, only works on single output function).

units:
Representative units of the physical quantity.

cell_methods:
List of blank-separated words of the form “name: method”. Must respect the CF-conventions and vocabulary (not checked).

description:
Sentence(s) meant to clarify the qualifiers of the fundamental quantities, such as which surface a quantity is defined on or what the flux sign conventions are.

comment:
Miscellaneous information about the data or methods used to produce it.

`cfcheck(`***das*`)`

Compare metadata attributes to CF-Convention standards.

Default cfchecks use the specifications in *xclim.core.utils.VARIABLES*, assuming the indicator's inputs are using the CMIP6/xclim variable names correctly. Variables absent from these default specs are silently ignored.

When subclassing this method, use functions decorated using *xclim.core.options.cfcheck*.

`static compute(`**args*, ***kwds*`)`

Compute the indicator.

This would typically be a function from *xclim.indices*.

`context = 'none'`

`datacheck(`***das*`)`

Verify that input data is valid.

When subclassing this method, use functions decorated using *xclim.core.options.datacheck*.

For example, checks could include:

- assert no precipitation is negative
- assert no temperature has the same value 5 days in a row

This base datacheck checks that the input data has a valid sampling frequency, as given in self.src_freq.

`classmethod from_dict(`*data: dict*, *identifier: str*, *module: str | None = None*`)`

Create an indicator subclass and instance from a dictionary of parameters.

Most parameters are passed directly as keyword arguments to the class constructor, except:

- "base" : A subclass of Indicator or a name of one listed in `xclim.core.indicator.registry` or `xclim.core.indicaotr.base_registry`. When passed, it acts as if *from_dict* was called on that class instead.
- "compute" : A string function name translates to a `xclim.indices.generic` or `xclim.indices` function.

Parameters

- **data** (*dict*) – The exact structure of this dictionary is detailed in the submodule documentation.
- **identifier** (*str*) – The name of the subclass and internal indicator name.
- **module** (*str*) – The module name of the indicator. This is meant to be used only if the indicator is part of a dynamically generated submodule, to override the module of the base class.

`identifier = None`

`property injected_parameters`

Return a dictionary of all injected parameters.

Opposite of `Indicator.parameters()`.

`json(`*args=None*`)`

Return a serializable dictionary representation of the class.

Parameters

args (*mapping, optional*) – Arguments as passed to the call method of the indicator. If not given, the default arguments will be used when formatting the attributes.

Notes

This is meant to be used by a third-party library wanting to wrap this class into another interface.

`keywords = ''`

`property n_outs`

Return the length of all cf_attrs.

`notes = ''`

`property parameters`

Create a dictionary of controllable parameters.

Similar to *`Indicator._all_parameters`*, but doesn't include injected parameters.

`realm = None`

`references = ''`

`src_freq = None`

`title = ''`

`classmethod translate_attrs(`*locale: str | Sequence[str], fill_missing: bool = True*`)`

Return a dictionary of unformated translated translatable attributes.

Translatable attributes are defined in *`xclim.core.locales.TRANSLATABLE_ATTRS`*.

Parameters

- **locale** (*Union[str, Sequence[str]]*) – The POSIX name of the locale or a tuple of a locale name and a path to a json file defining the translations. See *xclim.locale* for details.
- **fill_missing** (*bool*) – If True (default) fill the missing attributes by their english values.

`class xclim.core.indicator.IndicatorRegistrar`

Bases: `object`

Climate Indicator registering object.

`classmethod get_instance()`

Return first found instance.

Raises *ValueError* if no instance exists.

`class xclim.core.indicator.Parameter(`*kind: ~xclim.core.utils.InputKind, default: ~typing.Any, description: str = '', units: str = <class 'xclim.core.indicator._empty'>, choices: set = <class 'xclim.core.indicator._empty'>, value: ~typing.Any = <class 'xclim.core.indicator._empty'>*`)`

Bases: `object`

Class for storing an indicator's controllable parameter.

For retrocompatibility, this class implements a "getitem" and a special "contains".

Example

```
>>> p = Parameter(InputKind.NUMBER, default=2, description="A simple number")
>>> p.units is Parameter._empty  # has not been set
True
>>> "units" in p  # Easier/retrocompatible way to test if units are set
False
>>> p.description
'A simple number'
>>> p["description"]  # Same as above, for convenience.
'A simple number'
```

`class _empty`

Bases: `object`

`asdict()` → dict

Format indicators as a dictionary.

`choices`

alias of *_empty*

`default`

alias of `_empty`

`description: str = ''`

`property injected: bool`

Indicate whether values are injected.

`classmethod is_parameter_dict`(*other: dict*) → bool

Return whether indicator has a parameter dictionary.

`kind:` *InputKind*

`units`

alias of *_empty*

`update`(*other: dict*) → None

Update a parameter's values from a dict.

`value`

alias of *_empty*

`class xclim.core.indicator.ResamplingIndicator`(***kwds*)

Bases: *Indicator*

Indicator that performs a resampling computation.

Compared to the base Indicator, this adds the handling of missing data, and the check of allowed periods.

Parameters

- **missing** (*{any, wmo, pct, at_least_n, skip, from_context}*) – The name of the missing value method. See *xclim.core.missing.MissingBase* to create new custom methods. If None, this will be determined by the global configuration (see *xclim.set_options*). Defaults to "from_context".

- **missing_options** (*dict, None*) – Arguments to pass to the *missing* function. If None, this will be determined by the global configuration.
- **allowed_periods** (*Sequence[str], optional*) – A list of allowed periods, i.e. base parts of the *freq* parameter. For example, indicators meant to be computed annually only will have *allowed_periods=["A"]*. *None* means "any period" or that the indicator doesn't take a *freq* argument.

`classmethod _ensure_correct_parameters(`*parameters*`)`

Ensure the parameters are correctly set and ordered.

Sets the correct variable default to be sure.

`_history_string(`***kwargs*`)`

`_postprocess(`*outs*, *das*, *params*`)`

Masking of missing values.

`_preprocess_and_checks(`*das*, *params*`)`

Perform parent's checks and also check if freq is allowed.

`allowed_periods = None`

`missing = 'from_context'`

`missing_options = None`

`class xclim.core.indicator.ResamplingIndicatorWithIndexing(`***kwds*`)`

Bases: *ResamplingIndicator*

Resampling indicator that also injects "indexer" kwargs to subset the inputs before computation.

`classmethod _injected_parameters()`

Create a list of tuples for arguments to inject, (name, Parameter).

`_preprocess_and_checks(`*das: dict[str, DataArray]*, *params: dict[str, Any]*`)`

Perform parent's checks and also check if freq is allowed.

`class xclim.core.indicator._empty`

Bases: `object`

`xclim.core.indicator.add_iter_indicators(`*module*`)`

Create an iterable of loaded indicators.

`xclim.core.indicator.build_indicator_module(`*name: str*, *objs: Mapping[str, Indicator]*, *doc: str | None = None*`)` → ModuleType

Create or update a module from imported objects.

The module is inserted as a submodule of *xclim.indicators*.

Parameters

- **name** (*str*) – New module name. If it already exists, the module is extended with the passed objects, overwriting those with same names.
- **objs** (*dict*) – Mapping of the indicators to put in the new module. Keyed by the name they will take in that module.
- **doc** (*str*) – Docstring of the new module. Defaults to a simple header. Invalid if the module already exists.

Returns
ModuleType – A indicator module built from a mapping of Indicators.

`xclim.core.indicator.build_indicator_module_from_yaml`(*filename: PathLike, name: str | None = None, indices: Mapping[str, Callable] | ModuleType | PathLike | None = None, translations: dict[str, dict | PathLike] | None = None, mode: str = 'raise', encoding: str = 'UTF8'*) → ModuleType

Build or extend an indicator module from a YAML file.

The module is inserted as a submodule of `xclim.indicators`. When given only a base filename (no 'yml' extesion), this tries to find custom indices in a module of the same name (*.py) and translations in json files (.*<lang>.json), see Notes.

Parameters

- **filename** (*PathLike*) – Path to a YAML file or to the stem of all module files. See Notes for behaviour when passing a basename only.
- **name** (*str, optional*) – The name of the new or existing module, defaults to the basename of the file. (e.g: *atmos.yml* -> *atmos*)
- **indices** (*Mapping of callables or module or path, optional*) – A mapping or module of indice functions or a python file declaring such a file. When creating the indicator, the name in the *index_function* field is first sought here, then the indicator class will search in xclim.indices.generic and finally in xclim.indices.
- **translations** (*Mapping of dicts or path, optional*) – Translated metadata for the new indicators. Keys of the mapping must be 2-char language tags. Values can be translations dictionaries as defined in *Internationalization*. They can also be a path to a json file defining the translations.
- **mode** (*{'raise', 'warn', 'ignore'}*) – How to deal with broken indice definitions.
- **encoding** (*str*) – The encoding used to open the *.yaml* and *.json* files. It defaults to UTF-8, overriding python's mechanism which is machine dependent.

Returns
ModuleType – A submodule of *pym:mod:`xclim.indicators.*

Notes

When the given *filename* has no suffix (usually '.yaml' or '.yml'), the function will try to load custom indice definitions from a file with the same name but with a *.py* extension. Similarly, it will try to load translations in **.<lang>.json* files, where *<lang>* is the IETF language tag.

For example. a set of custom indicators could be fully described by the following files:

- *example.yml* : defining the indicator's metadata.
- *example.py* : defining a few indice functions.
- *example.fr.json* : French translations
- *example.tlh.json* : Klingon translations.

See also:

`The`

xclim.core.locales module

Internationalization

This module defines methods and object to help the internationalization of metadata for climate indicators computed by xclim. Go to *Adding translated metadata* to see how to use this feature.

All the methods and objects in this module use localization data given in json files. These files are expected to be defined as in this example for french:

```
{
    "attrs_mapping": {
        "modifiers": ["", "f", "mpl", "fpl"],
        "YS": ["annuel", "annuelle", "annuels", "annuelles"],
        "AS-*": ["annuel", "annuelle", "annuels", "annuelles"],
        # ... and so on for other frequent parameters translation...
    },
    "DTRVAR": {
        "long_name": "Variabilité de l'amplitude de la température diurne",
        "description": "Variabilité {freq:f} de l'amplitude de la température diurne␣
↪(définie comme la moyenne de la variation journalière de l'amplitude de température␣
↪sur une période donnée)",
        "title": "Variation quotidienne absolue moyenne de l'amplitude de la température␣
↪diurne",
        "comment": "",
        "abstract": "La valeur absolue de la moyenne de l'amplitude de la température␣
↪diurne.",
    },
    # ... and so on for other indicators...
}
```

Indicators are named by subclass identifier, the same as in the indicator registry (*xclim.core.indicators.registry*), but which can differ from the callable name. In this case, the indicator is called through *atmos.daily_temperature_range_variability*, but its identifier is *DTRVAR*. Use the *ind.__class__.__name__* accessor to get its registry name.

Here, the usual parameter passed to the formatting of "description" is "freq" and is usually translated from "YS" to "annual". However, in french and in this sentence, the feminine form should be used, so the "f" modifier is added by the translator so that the formatting function knows which translation to use. Acceptable entries for the mappings are limited to what is already defined in *xclim.core.indicators.utils.default_formatter*.

For user-provided internationalization dictionaries, only the "attrs_mapping" and its "modifiers" key are mandatory, all other entries (translations of frequent parameters and all indicator entries) are optional. For xclim-provided translations (for now only french), all indicators must have en entry and the "attrs_mapping" entries must match exactly the default formatter. Those default translations are found in the *xclim/locales* folder.

`xclim.core.locales.TRANSLATABLE_ATTRS = ['long_name', 'description', 'comment', 'title', 'abstract', 'keywords']`

List of attributes to consider translatable when generating locale dictionaries.

`exception xclim.core.locales.UnavailableLocaleError(`*locale*`)`

Bases: `ValueError`

Error raised when a locale is requested but doesn't exist.

`xclim.core.locales._valid_locales`(*locales*)

Check if the locales are valid.

`xclim.core.locales.generate_local_dict`(*locale: str*, *init_english: bool = False*) → dict

Generate a dictionary with keys for each indicator and translatable attributes.

Parameters

- **locale** (*str*) – Locale in the IETF format
- **init_english** (*bool*) – If True, fills the initial dictionary with the english versions of the attributes. Defaults to False.

`xclim.core.locales.get_local_attrs`(*indicator: str | Sequence[str]*, **locales: str | Sequence[str] | tuple[str, dict]*, *names: Sequence[str] | None = None*, *append_locale_name: bool = True*) → dict

Get all attributes of an indicator in the requested locales.

Parameters

- **indicator** (*str or sequence of strings*) – Indicator's class name, usually the same as in *xc.core.indicator.registry*. If multiple names are passed, the attrs from each indicator are merged, with the highest priority set to the first name.
- **locales** (*str or tuple of str*) – IETF language tag or a tuple of the language tag and a translation dict, or a tuple of the language tag and a path to a json file defining translation of attributes.
- **names** (*Optional[Sequence[str]]*) – If given, only returns translations of attributes in this list.
- **append_locale_name** (*bool*) – If True (default), append the language tag (as "{attr_name}_{locale}") to the returned attributes.

Raises

`ValueError` – If *append_locale_name* is False and multiple *locales* are requested.

Returns

dict – All CF attributes available for given indicator and locales. Warns and returns an empty dict if none were available.

`xclim.core.locales.get_local_dict`(*locale: str | Sequence[str] | tuple[str, dict]*) → tuple[str, dict]

Return all translated metadata for a given locale.

Parameters

locale (*str or sequence of str*) – IETF language tag or a tuple of the language tag and a translation dict, or a tuple of the language tag and a path to a json file defining translation of attributes.

Raises

`UnavailableLocaleError` – If the given locale is not available.

Returns

- *str* – The best fitting locale string
- *dict* – The available translations in this locale.

`xclim.core.locales.get_local_formatter`(*locale: str | Sequence[str] | tuple[str, dict]*) → *AttrFormatter*

Return an AttrFormatter instance for the given locale.

Parameters
locale (*str or tuple of str*) – IETF language tag or a tuple of the language tag and a translation dict, or a tuple of the language tag and a path to a json file defining translation of attributes.

`xclim.core.locales.list_locales()`

List of loaded locales. Includes all loaded locales, no matter how complete the translations are.

`xclim.core.locales.load_locale(`*locdata: str | Path | Mapping[str, dict]*, *locale: str*`)`

Load translations from a json file into xclim.

Parameters

- **locdata** (*str or dictionary*) – Either a loaded locale dictionary or a path to a json file.
- **locale** (*str*) – The locale name (IETF tag).

`xclim.core.locales.read_locale_file(`*filename*, *module: str | None = None*, *encoding: str = 'UTF8'*`)` → dict

Read a locale file (.json) and return its dictionary.

Parameters

- **filename** (*PathLike*) – The file to read.
- **module** (*str, optional*) – If module is a string, this module name is added to all identifiers translated in this file. Defaults to None, and no module name is added (as if the indicator was an official xclim indicator).
- **encoding** (*str*) – The encoding to use when reading the file. Defaults to UTF-8, overriding python's default mechanism which is machine dependent.

xclim.core.missing module

Missing values identification

Indicators may use different criteria to determine whether a computed indicator value should be considered missing. In some cases, the presence of any missing value in the input time series should result in a missing indicator value for that period. In other cases, a minimum number of valid values or a percentage of missing values should be enforced. The World Meteorological Organisation (WMO) suggests criteria based on the number of consecutive and overall missing values per month.

xclim has a registry of missing value detection algorithms that can be extended by users to customize the behavior of indicators. Once registered, algorithms can be used within indicators by setting the *missing* attribute of an *Indicator* subclass. By default, *xclim* registers the following algorithms:

- *any*: A result is missing if any input value is missing.
- *at_least_n*: A result is missing if less than a given number of valid values are present.
- *pct*: A result is missing if more than a given fraction of values are missing.
- *wmo*: A result is missing if 11 days are missing, or 5 consecutive values are missing in a month.
- *skip*: Skip missing value detection.
- *from_context*: Look-up the missing value algorithm from options settings. See `xclim.set_options()`.

To define another missing value algorithm, subclass `MissingBase` and decorate it with *`xclim.core.options.register_missing_method()`*.

`xclim.core.missing.at_least_n_valid(`*da*, *freq*, *n=1*, *src_timestep=None*, ***indexer*`)`

Return whether there are at least a given number of valid values.

Parameters

- **da** (*DataArray*) – Input array.
- **freq** (*str*) – Resampling frequency.
- **n** (*int*) – Minimum of valid values required.
- **src_timestep** (*{"D", "H"}*) – Expected input frequency.
- **indexer** (*{dim: indexer, }, optional*) – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered.

Returns

out (*DataArray*) – A boolean array set to True if period has missing values.

`xclim.core.missing.missing_any(`*da*, *freq*, *src_timestep=None*, ***indexer*`)`

Return whether there are missing days in the array.

Parameters

- **da** (*DataArray*) – Input array.
- **freq** (*str*) – Resampling frequency.
- **src_timestep** (*{"D", "H", "M"}*) – Expected input frequency.
- **indexer** (*{dim: indexer, }, optional*) – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered.

Returns

DataArray – A boolean array set to True if period has missing values.

`xclim.core.missing.missing_from_context(`*da*, *freq*, *src_timestep=None*, ***indexer*`)`

Return whether each element of the resampled da should be considered missing according to the currently set options in *xclim.set_options*.

See *xclim.set_options* and *xclim.core.options.register_missing_method*.

`xclim.core.missing.missing_pct(`*da*, *freq*, *tolerance*, *src_timestep=None*, ***indexer*`)`

Return whether there are more missing days in the array than a given percentage.

Parameters

- **da** (*DataArray*) – Input array.
- **freq** (*str*) – Resampling frequency.
- **tolerance** (*float*) – Fraction of missing values that are tolerated [0,1].
- **src_timestep** (*{"D", "H"}*) – Expected input frequency.
- **indexer** (*{dim: indexer, }, optional*) – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered.

Returns
DataArray – A boolean array set to True if period has missing values.

`xclim.core.missing.missing_wmo(`*da*, *freq*, *nm=11*, *nc=5*, *src_timestep=None*, ***indexer*`)`

Return whether a series fails WMO criteria for missing days.

The World Meteorological Organisation recommends that where monthly means are computed from daily values, it should be considered missing if either of these two criteria are met:

> – observations are missing for 11 or more days during the month; – observations are missing for a period of 5 or more consecutive days during the month.

Stricter criteria are sometimes used in practice, with a tolerance of 5 missing values or 3 consecutive missing values.

Parameters

- **da** (*DataArray*) – Input array.
- **freq** (*str*) – Resampling frequency.
- **nm** (*int*) – Number of missing values per month that should not be exceeded.
- **nc** (*int*) – Number of consecutive missing values per month that should not be exceeded.
- **src_timestep** (*{"D"}*) – Expected input frequency. Only daily values are supported.
- **indexer** (*{dim: indexer, }, optional*) – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered.

Returns
DataArray – A boolean array set to True if period has missing values.

Notes

If used at frequencies larger than a month, for example on an annual or seasonal basis, the function will return True if any month within a period is missing.

`xclim.core.missing.register_missing_method(`*name: str*`)` → Callable

Register missing method.

xclim.core.options module

Options submodule

Global or contextual options for xclim, similar to xarray.set_options.

`xclim.core.options._run_check(`*func*, *option*, **args*, ***kwargs*`)`

Run function and customize exception handling based on option.

`xclim.core.options._set_metadata_locales(`*locales*`)`

`xclim.core.options._set_missing_options(`*mopts*`)` `xclim.core.options._valid_missing_options(`*mopts*`)`

`xclim.core.options.cfcheck(`*func: Callable*`)` → Callable

Decorate functions checking CF-compliance of DataArray attributes.

Functions should raise ValidationError exceptions whenever attributes are non-conformant.

`xclim.core.options.datacheck(`*func: Callable*`)` → Callable

Decorate functions checking data inputs validity.

`xclim.core.options.register_missing_method(`*name: str*`)` → Callable

Register missing method.

`class xclim.core.options.set_options(`***kwargs*`)`

Bases: `object`

Set options for xclim in a controlled context.

Currently-supported options:

- `metadata_locales`: List of IETF language tags or tuples of language tags and a translation dict, or tuples of language tags and a path to a json file defining translation of attributes. Default: `[]`.
- `data_validation`: Whether to 'log', 'raise' an error or 'warn' the user on inputs that fail the data checks in *xclim.core.datachecks*. Default: `'raise'`.
- `cf_compliance`: Whether to 'log', 'raise' an error or 'warn' the user on inputs that fail the CF compliance checks in *xclim.core.cfchecks*. Default: `'warn'`.
- `check_missing`: How to check for missing data and flag computed indicators. Default available methods are "any", "wmo", "pct", "at_least_n" and "skip". Missing method can be registered through the *xclim.core.options.register_missing_method* decorator. Default: `'any'`
- `missing_options`: Dictionary of options to pass to the missing method. Keys must the name of missing method and values must be mappings from option names to values.
- `run_length_ufunc`: Whether to use the 1D ufunc version of run length algorithms or the dask-ready broadcasting version. Default is `'auto'` which means the latter is used for dask-backed and large arrays.
- `sdba_extra_output`: Whether to add diagnostic variables to outputs of sdba's *train*, *adjust* and *processing* operations. Details about these additional variables are given in the object's docstring. When activated, *adjust* will return a Dataset with *scen* and those extra diagnostics For *processing* functions, see the doc, the output type might change, or not depending on the algorithm. Default: `False`.
- `sdba_encode_cf`: Whether to encode cf coordinates in the `map_blocks` optimization that most adjustment methods are based on. This should have no impact on the results, but should run much faster in the graph creation phase.
- `keep_attrs`: Controls attributes handling in indicators. If True, attributes from all inputs are merged using the *drop_conflicts* strategy and then updated with xclim-provided attributes. If False, attributes from the inputs are ignored. If "xarray", xclim will use xarray's *keep_attrs* option. Note that xarray's "default" is equivalent to False. Default: `"xarray"`.

Examples

You can use `set_options` either as a context manager:

```
>>> import xclim
>>> ds = xr.open_dataset(path_to_tas_file).tas
>>> with xclim.set_options(metadata_locales=["fr"]):
...     out = xclim.atmos.tg_mean(ds)
...
```

Or to set global options:

```
>>> xclim.set_options(
...     missing_options={"pct": {"tolerance": 0.04}}
... )
<xclim.core.options.set_options object at ...>
```

`_update`(*kwargs*)
: Update values.

xclim.core.units module

Units handling submodule

Pint is used to define the *units UnitRegistry* and *xclim.units.core* defines most unit handling methods.

`xclim.core.units.check_units`(*val: str | int | float | None*, *dim: str | None*) → None

Check units for appropriate convention compliance.

`xclim.core.units.convert_units_to`(*source: str | xr.DataArray | Any*, *target: str | xr.DataArray | Any*, *context: str | None = None*) → xr.DataArray | float | int | str | Any

Convert a mathematical expression into a value with the same units as a DataArray.

Parameters

- **source** (*Union[str, xr.DataArray, Any]*) – The value to be converted, e.g. '4C' or '1 mm/d'.
- **target** (*Union[str, xr.DataArray, Any]*) – Target array of values to which units must conform.
- **context** (*str, optional*) – The unit definition context. Default: None.

Returns

Union[xr.DataArray, float, int, str, Any] – The source value converted to target's units.

`xclim.core.units.declare_units`(***units_by_name*) → Callable

Create a decorator to check units of function arguments.

The decorator checks that input and output values have units that are compatible with expected dimensions. It also stores the input units as a 'in_units' attribute.

Parameters

units_by_name (*Mapping[str, str]*) – Mapping from the input parameter names to their units or dimensionality ("[...]").

Examples

In the following function definition:

```
@declare_units(tas=["temperature"])
def func(tas):
    ...
```

The decorator will check that *tas* has units of temperature (C, K, F).

`xclim.core.units.infer_sampling_units(`*da: xr.DataArray, deffreq: str | None = 'D', dim: str = 'time'*`)` → tuple[int, str]

Infer a multiplicator and the units corresponding to one sampling period.

Parameters

- **da** (*xr.DataArray*) – A DataArray from which to take coordinate *dim*.
- **deffreq** (*str*) – If no frequency is inferred from *da[dim]*, take this one.
- **dim** (*str*) – Dimension from which to infer the frequency.

Raises

`ValueError` – If the frequency has no exact corresponding units.

Returns

- **m** (*int*) – The magnitude (number of base periods per period)
- **u** (*str*) – Units as a string, understandable by pint.

`xclim.core.units.pint2cfunits(`*value: UnitDefinition*`)` → str

Return a CF-compliant unit string from a *pint* unit.

Parameters

value (*pint.Unit*) – Input unit.

Returns

out (*str*) – Units following CF-Convention, using symbols.

`xclim.core.units.pint_multiply(`*da: xr.DataArray, q: Any, out_units: str | None = None*`)`

Multiply xarray.DataArray by pint.Quantity.

Parameters

- **da** (*xr.DataArray*) – Input array.
- **q** (*pint.Quantity*) – Multiplicative factor.
- **out_units** (*Optional[str]*) – Units the output array should be converted into.

`xclim.core.units.rate2amount(`*rate: DataArray, dim: str = 'time', out_units: Optional[str] = None*`)` → DataArray

Convert a rate variable to an amount by multiplying by the sampling period length.

If the sampling period length cannot be inferred, the rate values are multiplied by the duration between their time coordinate and the next one. The last period is estimated with the duration of the one just before.

This is the inverse operation of `amount2rate()`.

Parameters

- **rate** (*xr.DataArray*) – "Rate" variable, with units of "amount" per time. Ex: Precipitation in "mm / d".
- **dim** (*str*) – The time dimension.
- **out_units** (*str, optional*) – Output units to convert to.

Returns
xr.DataArray

Examples

The following converts a daily array of precipitation in mm/h to the daily amounts in mm.

```
>>> time = xr.cftime_range("2001-01-01", freq="D", periods=365)
>>> pr = xr.DataArray(
...     [1] * 365, dims=("time",), coords={"time": time}, attrs={"units": "mm/h"}
... )
>>> pram = rate2amount(pr)
>>> pram.units
'mm'
>>> float(pram[0])
24.0
```

Also works if the time axis is irregular : the rates are assumed constant for the whole period starting on the values timestamp to the next timestamp.

```
>>> time = time[[0, 9, 30]]  # The time axis is Jan 1st, Jan 10th, Jan 31st
>>> pr = xr.DataArray(
...     [1] * 3, dims=("time",), coords={"time": time}, attrs={"units": "mm/h"}
... )
>>> pram = rate2amount(pr)
>>> pram.values
array([216., 504., 504.])
```

Finally, we can force output units:

```
>>> pram = rate2amount(pr, out_units="pc")  # Get rain amount in parsecs. Why not.
>>> pram.values
array([7.00008327e-18, 1.63335276e-17, 1.63335276e-17])
```

`xclim.core.units.str2pint(`*val: str*`)`

Convert a string to a pint.Quantity, splitting the magnitude and the units.

Parameters
val (*str*) – A quantity in the form "[{magnitude}]{units}", where magnitude is castable to a float and units is understood by *units2pint*.

Returns
pint.Quantity – Magnitude is 1 if no magnitude was present in the string.

`xclim.core.units.to_agg_units(`*out: DataArray, orig: DataArray, op: str, dim: str = 'time'*`) → DataArray`

Set and convert units of an array after an aggregation operation along the sampling dimension (time).

Parameters

- **out** (*xr.DataArray*) – The output array of the aggregation operation, no units operation done yet.
- **orig** (*xr.DataArray*) – The original array before the aggregation operation, used to infer the sampling units and get the variable units.
- **op** (*{'count', 'prod', 'delta_prod'}*) – The type of aggregation operation performed. The special "delta_*" ops are used with temperature units needing conversion to their "delta" counterparts (e.g. degree days)
- **dim** (*str*) – The time dimension along which the aggregation was performed.

Examples

Take a daily array of temperature and count number of days above a threshold. *to_agg_units* will infer the units from the sampling rate along "time", so we ensure the final units are correct.

```
>>> time = xr.cftime_range("2001-01-01", freq="D", periods=365)
>>> tas = xr.DataArray(
...     np.arange(365),
...     dims=("time",),
...     coords={"time": time},
...     attrs={"units": "degC"},
... )
>>> cond = tas > 100  # Which days are boiling
>>> Ndays = cond.sum("time")  # Number of boiling days
>>> Ndays.attrs.get("units")
None
>>> Ndays = to_agg_units(Ndays, tas, op="count")
>>> Ndays.units
'd'
```

Similarly, here we compute the total heating degree-days but we have weekly data: >>> time = xr.cftime_range("2001-01-01", freq="7D", periods=52) >>> tas = xr.DataArray(... np.arange(52) + 10, ... dims=("time",), ... coords={"time": time}, ... attrs={"units": "degC"}, ...) >>> degdays = (... (tas - 16).clip(0).sum("time") ...) # Integral of temperature above a threshold >>> degdays = to_agg_units(degdays, tas, op="delta_prod") >>> degdays.units 'week delta_degC'

Which we can always convert to the more common "K days":

```
>>> degdays = convert_units_to(degdays, "K days")
>>> degdays.units
'K d'
```

`xclim.core.units.units2pint`(*value: xr.DataArray | str | units.Quantity*) → Unit

Return the pint Unit for the DataArray units.

Parameters

value (*Union[xr.DataArray, str. pint.Quantity]*) – Input data array or string representing a unit (with no magnitude).

Returns

pint.unit.UnitDefinition – Units of the data array.

xclim.core.utils module

Miscellaneous indices utilities

Helper functions for the indices computations, indicator construction and other things.

`xclim.core.utils.DateStr`

Type annotation for strings representing full dates (YYYY-MM-DD), may include time.

alias of `str`

`xclim.core.utils.DayOfYearStr`

Type annotation for strings representing dates without a year (MM-DD).

alias of `str`

`class xclim.core.utils.InputKind(`*value*`)`

Bases: `IntEnum`

Constants for input parameter kinds.

For use by external parses to determine what kind of data the indicator expects. On the creation of an indicator, the appropriate constant is stored in *xclim.core.indicator.Indicator.parameters*. The integer value is what gets stored in the output of *xclim.core.indicator.Indicator.json()*.

For developers : for each constant, the docstring specifies the annotation a parameter of an indice function should use in order to be picked up by the indicator constructor. Notice that we are using the annotation format as described in PEP604/py3.10, i.e. with | indicating an union and without import objects from *typing*.

`BOOL = 9`

A boolean flag.

Annotation : `bool`, may be optional.

`DATASET = 70`

An xarray dataset.

Developers : as indices only accept DataArrays, this should only be added on the indicator's constructor.

`DATE = 7`

A date in the YYYY-MM-DD format, may include a time.

Annotation : *xclim.core.utils.DateStr* (may be optional).

`DAY_OF_YEAR = 6`

A date, but without a year, in the MM-DD format.

Annotation : *xclim.core.utils.DayOfYearStr* (may be optional).

`FREQ_STR = 3`

A string representing an "offset alias", as defined by pandas.

See https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases . Annotation : `str` + `freq` as the parameter name.

`KWARGS = 50`

A mapping from argument name to value.

Developers : maps the `**kwargs`. Please use as little as possible.

`NUMBER = 4`

A number.

Annotation : `int`, `float` and unions thereof, potentially optional.

`NUMBER_SEQUENCE = 8`

A sequence of numbers

Annotation : `Sequence[int]`, `Sequence[float]` and unions thereof, may include single `int` and `float`, may be optional.

`OPTIONAL_VARIABLE = 1`

An optional data variable (DataArray or variable name).

Annotation : `xr.DataArray | None`. The default should be None.

`OTHER_PARAMETER = 99`

An object that fits None of the previous kinds.

Developers : This is the fallback kind, it will raise an error in xclim's unit tests if used.

`QUANTITY_STR = 2`

A string representing a quantity with units.

Annotation : `str` + an entry in the `xclim.core.units.declare_units()` decorator.

`STRING = 5`

A simple string.

Annotation : `str` or `str | None`. In most cases, this kind of parameter makes sense with choices indicated in the docstring's version of the annotation with curly braces. See *Defining new indices*.

`VARIABLE = 0`

A data variable (DataArray or variable name).

Annotation : `xr.DataArray`.

`exception xclim.core.utils.MissingVariableError`

Bases: `ValueError`

Error raised when a dataset is passed to an indicator but one of the needed variable is missing.

`class xclim.core.utils.PercentileDataArray(`*data: Any = <NA>, coords: Sequence[tuple] | Mapping[Any, Any] | None = None, dims: Hashable | Sequence[Hashable] | None = None, name: Hashable = None, attrs: Mapping = None, indexes: dict[Hashable, pd.Index] = None, fastpath: bool = False*`)`

Bases: `DataArray`

Wrap xarray DataArray for percentiles values.

This class is used internally with its corresponding InputKind to recognize this sort of input and to retrieve from it the attributes needed to build indicator metadata.

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

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

Parameters

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

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

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

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

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

Parameters

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

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

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

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

Parameters

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

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

`cumprod`(*dim=None*, *axis=None*, *skipna=None*, ***kwargs*)

Apply *cumprod* along some dimension of PercentileDataArray.

Parameters

- **dim** (*str or sequence of str, optional*) – Dimension over which to apply *cumprod*.
- **axis** (*int or sequence of int, optional*) – Axis over which to apply *cumprod*. Only one of the 'dim' and 'axis' arguments can be supplied.
- **skipna** (*bool, optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **keep_attrs** (*bool, optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- ****kwargs** (*dict*) – Additional keyword arguments passed on to *cumprod*.

Returns

cumvalue (*PercentileDataArray*) – New PercentileDataArray object with *cumprod* applied to its data along the indicated dimension.

`cumsum(`*dim=None*, *axis=None*, *skipna=None*, ***kwargs*`)`

Apply *cumsum* along some dimension of PercentileDataArray.

Parameters

- **dim** (*str or sequence of str, optional*) – Dimension over which to apply *cumsum*.
- **axis** (*int or sequence of int, optional*) – Axis over which to apply *cumsum*. Only one of the 'dim' and 'axis' arguments can be supplied.
- **skipna** (*bool, optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **keep_attrs** (*bool, optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- ****kwargs** (*dict*) – Additional keyword arguments passed on to *cumsum*.

Returns

cumvalue (*PercentileDataArray*) – New PercentileDataArray object with *cumsum* applied to its data along the indicated dimension.

`classmethod from_da(`*source: xr.DataArray*, *climatology_bounds: list[str] = None*`)` → *PercentileDataArray*

Create a PercentileDataArray from a xarray.DataArray.

Parameters

- **source** (*DataArray*) – A DataArray with its content containing percentiles values. It must also have a coordinate variable percentiles or quantile.
- **climatology_bounds** (*list[str]*) – Optional. A List of size two which contains the period on which the percentiles were computed. See *xclim.core.calendar.build_climatology_bounds* to build this list from a DataArray.

Returns

PercentileDataArray – The initial *source* DataArray but wrap by PercentileDataArray class. The data is unchanged and only climatology_bounds attributes is overridden if q new value is given in inputs.

`classmethod is_compatible(`*source: DataArray*`)` → bool

Evaluate whether PecentileDataArray is conformant with expected fields.

A PercentileDataArray must have climatology_bounds attributes and either a quantile or percentiles coordinate, the window is not mandatory.

`item(`**args*`)`

Copy an element of an array to a standard Python scalar and return it.

Parameters

***args** (*Arguments (variable number and type)*) –

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z (*Standard Python scalar object*) – A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

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

Reduce this PercentileDataArray's data by applying *max* along some dimension(s).

Parameters

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

Returns

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

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

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

Parameters

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

Returns

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

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

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

Parameters

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

a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).

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

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

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

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

Parameters

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

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

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

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

Parameters

- **dim** (*str or sequence of str, optional*) – Dimension(s) over which to apply *prod*.
- **axis** (*int or sequence of int, optional*) – Axis(es) over which to apply *prod*. Only one of the 'dim' and 'axis' arguments can be supplied. If neither are supplied, then *prod* is calculated over axes.
- **skipna** (*bool, optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **min_count** (*int, default: None*) – The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA. Only used if skipna is set to True or defaults to True for the array's dtype. New in version 0.10.8: Added with the default being None. Changed in version

0.17.0: if specified on an integer array and skipna=True, the result will be a float array.

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

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

`searchsorted`(*v*, *side='left'*, *sorter=None*)

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see *numpy.searchsorted*

See also:

`numpy.searchsorted`
equivalent function

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

Reduce this PercentileDataArray's data by applying *std* along some dimension(s).

Parameters

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

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

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

Reduce this PercentileDataArray's data by applying *sum* along some dimension(s).

Parameters

- **dim** (*str or sequence of str, optional*) – Dimension(s) over which to apply *sum*.
- **axis** (*int or sequence of int, optional*) – Axis(es) over which to apply *sum*. Only one of the 'dim' and 'axis' arguments can be supplied. If neither are supplied, then *sum* is calculated over axes.

- **skipna** (*bool, optional*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).
- **min_count** (*int, default: None*) – The required number of valid values to perform the operation. If fewer than min_count non-NA values are present the result will be NA. Only used if skipna is set to True or defaults to True for the array's dtype. New in version 0.10.8: Added with the default being None. Changed in version 0.17.0: if specified on an integer array and skipna=True, the result will be a float array.
- **keep_attrs** (*bool, optional*) – If True, the attributes (*attrs*) will be copied from the original object to the new one. If False (default), the new object will be returned without attributes.
- ****kwargs** (*dict*) – Additional keyword arguments passed on to the appropriate array function for calculating *sum* on this object's data.

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

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

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

Parameters

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

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

`exception xclim.core.utils.ValidationError`

Bases: `ValueError`

Error raised when input data to an indicator fails the validation tests.

`property msg`

`xclim.core.utils._compute_virtual_index`(*n: ndarray*, *quantiles: ndarray*, *alpha: float*, *beta: float*)

Compute the floating point indexes of an array for the linear interpolation of quantiles.

Based on the approach used by [Hyndman&Fan1996]_.

Parameters

- **n** (*array_like*) – The sample sizes.
- **quantiles** (*array_like*) – The quantiles values.
- **alpha** (*float*) – A constant used to correct the index computed.
- **beta** (*float*) – A constant used to correct the index computed.

Notes

alpha and *beta* values depend on the chosen method (see quantile documentation).

References

`xclim.core.utils._get_gamma`(*virtual_indexes: ndarray*, *previous_indexes: ndarray*)

Compute gamma (AKA 'm' or 'weight') for the linear interpolation of quantiles.

Parameters

- **virtual_indexes** (*array_like*) – The indexes where the percentile is supposed to be found in the sorted sample.
- **previous_indexes** (*array_like*) – The floor values of virtual_indexes.

Notes

gamma is usually the fractional part of virtual_indexes but can be modified by the interpolation method.

`xclim.core.utils._get_indexes`(*arr: np.ndarray*, *virtual_indexes: np.ndarray*, *valid_values_count: np.ndarray*) → tuple[np.ndarray, np.ndarray]

Get the valid indexes of arr neighbouring virtual_indexes.

Notes

This is a companion function to linear interpolation of quantiles.

Returns

array-like, array-like – A tuple of virtual_indexes neighbouring indexes (previous and next)

`xclim.core.utils._linear_interpolation`(*left: ndarray*, *right: ndarray*, *gamma: ndarray*) → ndarray

Compute the linear interpolation weighted by gamma on each point of two same shape arrays.

Parameters

- **left** (*array_like*) – Left bound.
- **right** (*array_like*) – Right bound.
- **gamma** (*array_like*) – The interpolation weight.

Returns

array_like

`xclim.core.utils._nan_quantile`(*arr: np.ndarray*, *quantiles: np.ndarray*, *axis: int = 0*, *alpha: float = 1.0*, *beta: float = 1.0*) → float | np.ndarray

Get the quantiles of the array for the given axis.

A linear interpolation is performed using alpha and beta.

Notes

By default, alpha == beta == 1 which performs the 7th method of [Hyndman&Fan1996]_. with alpha == beta == 1/3 we get the 8th method.

`xclim.core.utils.adapt_clix_meta_yaml`(*raw: PathLike*, *adapted: PathLike*)

Read in a clix-meta yaml and refactor it to fit xclim's yaml specifications.

`xclim.core.utils.calc_perc`(*arr: ndarray*, *percentiles: Optional[Sequence[float]] = None*, *alpha: float = 1.0*, *beta: float = 1.0*, *copy: bool = True*) → ndarray

Compute percentiles using nan_calc_percentiles and move the percentiles' axis to the end.

`xclim.core.utils.ensure_chunk_size`(*da: DataArray*, ***minchunks: Mapping[str, int]*) → DataArray

Ensure that the input DataArray has chunks of at least the given size.

If only one chunk is too small, it is merged with an adjacent chunk. If many chunks are too small, they are grouped together by merging adjacent chunks.

Parameters

- **da** (*xr.DataArray*) – The input DataArray, with or without the dask backend. Does nothing when passed a non-dask array.
- **minchunks** (*Mapping[str, int]*) – A kwarg mapping from dimension name to minimum chunk size. Pass -1 to force a single chunk along that dimension.

`xclim.core.utils.infer_kind_from_parameter`(*param: Parameter*, *has_units: bool = False*) → *InputKind*

Return the appropriate InputKind constant from an `inspect.Parameter` object.

The correspondance between parameters and kinds is documented in `xclim.core.utils.InputKind`. The only information not inferable through the *inspect* object is whether the parameter has been assigned units through the `xclim.core.units.declare_units()` decorator. That can be given with the `has_units` flag.

`xclim.core.utils.load_module`(*path: os.PathLike*, *name: str | None = None*)

Load a python module from a python file, optionally changing its name.

Examples

Given a path to a module file (.py)

```
>>> # xdoctest: +SKIP
>>> from pathlib import Path
>>> path = Path("path/to/example.py")
```

The two following imports are equivalent, the second uses this method.

```
>>> os.chdir(path.parent)
>>> import example as mod1
>>> os.chdir(previous_working_dir)
>>> mod2 = load_module(path)
>>> mod1 == mod2
```

`xclim.core.utils.nan_calc_percentiles`(*arr: ndarray*, *percentiles: Optional[Sequence[float]] = None*, *axis=- 1*, *alpha=1.0*, *beta=1.0*, *copy=True*) → ndarray

Convert the percentiles to quantiles and compute them using _nan_quantile.

`xclim.core.utils.raise_warn_or_log`(*err: Exception*, *mode: str*, *msg: str | None = None*, *err_type=<class 'ValueError'>*, *stacklevel: int = 1*)

Raise, warn or log an error according.

Parameters

- **err** (*Exception*) – An error.
- **mode** (*{'ignore', 'log', 'warn', 'raise'}*) – What to do with the error.
- **msg** (*str, optional*) – The string used when logging or warning. Defaults to the *msg* attr of the error (if present) or to "Failed with <err>".
- **err_type** (*type*) – The type of error/exception to raise.
- **stacklevel** (*int*) – Stacklevel when warning. Relative to the call of this function (1 is added).

`xclim.core.utils.uses_dask`(*da*)

Evaluate whether dask is installed and array is loaded as a dask array.

`xclim.core.utils.walk_map`(*d: dict*, *func: function*) → dict

Apply a function recursively to values of dictionary.

Parameters

- **d** (*dict*) – Input dictionary, possibly nested.
- **func** (*FunctionType*) – Function to apply to dictionary values.

Returns

dict – Dictionary whose values are the output of the given function.

`xclim.core.utils.wrapped_partial`(*func: FunctionType*, *suggested: dict | None = None*, ***fixed*) → Callable

Wrap a function, updating its signature but keeping its docstring.

Parameters

- **func** (*FunctionType*) – The function to be wrapped
- **suggested** (*dict*) – Keyword arguments that should have new default values but still appear in the signature.
- **fixed** (*kwargs*) – Keyword arguments that should be fixed by the wrapped and removed from the signature.

Examples

```
>>> from inspect import signature
>>> def func(a, b=1, c=1):
...     print(a, b, c)
...
>>> newf = wrapped_partial(func, b=2)
>>> signature(newf)
<Signature (a, *, c=1)>
>>> newf(1)
1 2 1
>>> newf = wrapped_partial(func, suggested=dict(c=2), b=2)
>>> signature(newf)
<Signature (a, *, c=2)>
>>> newf(1)
1 2 2
```

xclim.data package

JSON and YAML definitions for virtual modules and internationalisation support.

xclim.ensembles package

Ensemble tools.

This submodule defines some useful methods for dealing with ensembles of climate simulations. In xclim, an "ensemble" is a *Dataset* or a *DataArray* where multiple climate realizations or models are concatenated along the *realization* dimension.

Submodules

xclim.ensembles._base module

Ensembles Creation and Statistics

`xclim.ensembles._base._ens_align_datasets`(*datasets: list[xr.Dataset | Path | str | list[Path | str]] | str, mf_flag: bool = False, resample_freq: str | None = None, calendar: str = 'default', **xr_kwargs*) → list[xr.Dataset]

Create a list of aligned xarray Datasets for ensemble Dataset creation.

Parameters

- **datasets** (*list[xr.Dataset | xr.DataArray | Path | str | list[Path | str]] or str*) – List of netcdf file paths or xarray Dataset/DataArray objects . If mf_flag is True, 'datasets' should be a list of lists where each sublist contains input NetCDF files of a xarray multi-file Dataset. DataArrays should have a name, so they can be converted to datasets. If a string, it is assumed to be a glob pattern for finding datasets.
- **mf_flag** (*bool*) – If True climate simulations are treated as xarray multi-file datasets before concatenation. Only applicable when 'datasets' is a sequence of file paths.

- **resample_freq** (*str or None*) – If the members of the ensemble have the same frequency but not the same offset, they cannot be properly aligned. If resample_freq is set, the time coordinate of each member will be modified to fit this frequency.
- **calendar** (*str*) – The calendar of the time coordinate of the ensemble. For conversions involving '360_day', the align_on='date' option is used. See *xclim.core.calendar.convert_calendar*. 'default' is the standard calendar using np.datetime64 objects.
- **xr_kwargs** – Any keyword arguments to be given to xarray when opening the files.

Returns
list[xr.Dataset]

`xclim.ensembles._base.create_ensemble(`*datasets: list[xr.Dataset | xr.DataArray | Path | str | list[Path | str]] | str*, *mf_flag: bool = False*, *resample_freq: str | None = None*, *calendar: str = 'default'*, ***xr_kwargs*`)` → xr.Dataset

Create an xarray dataset of an ensemble of climate simulation from a list of netcdf files.

Input data is concatenated along a newly created data dimension ('realization'). Returns an xarray dataset object containing input data from the list of netcdf files concatenated along a new dimension (name:'realization'). In the case where input files have unequal time dimensions, the output ensemble Dataset is created for maximum time-step interval of all input files. Before concatenation, datasets not covering the entire time span have their data padded with NaN values. Dataset and variable attributes of the first dataset are copied to the resulting dataset.

Parameters

- **datasets** (*List[Union[xr.Dataset, Path, str, List[Path, str]]] or str*) – List of netcdf file paths or xarray Dataset/DataArray objects . If mf_flag is True, ncfiles should be a list of lists where each sublist contains input .nc files of an xarray multifile Dataset. If DataArray object are passed, they should have a name in order to be transformed into Datasets. If a string is passed, it is assumed to be a glob pattern for finding datasets.
- **mf_flag** (*bool*) – If True, climate simulations are treated as xarray multifile Datasets before concatenation. Only applicable when "datasets" is a sequence of file paths.
- **resample_freq** (*Optional[str]*) – If the members of the ensemble have the same frequency but not the same offset, they cannot be properly aligned. If resample_freq is set, the time coordinate of each members will be modified to fit this frequency.
- **calendar** (*str*) – The calendar of the time coordinate of the ensemble. For conversions involving '360_day', the align_on='date' option is used. See *xclim.core.calendar.convert_calendar*. 'default' is the standard calendar using np.datetime64 objects.
- **xr_kwargs** – Any keyword arguments to be given to *xr.open_dataset* when opening the files (or to *xr.open_mfdataset* if mf_flag is True)

Returns
xr.Dataset – Dataset containing concatenated data from all input files.

Notes

Input netcdf files require equal spatial dimension size (e.g. lon, lat dimensions). If input data contains multiple cftime calendar types they must be at monthly or coarser frequency.

Examples

```
>>> from xclim.ensembles import create_ensemble
>>> ens = create_ensemble(temperature_datasets)
```

Using multifile datasets, through glob patterns. Simulation 1 is a list of .nc files (e.g. separated by time):

```
>>> datasets = glob.glob("/dir/*.nc")
```

Simulation 2 is also a list of .nc files:

```
>>> datasets.append(glob.glob("/dir2/*.nc"))
>>> ens = create_ensemble(datasets, mf_flag=True)
```

`xclim.ensembles._base.ensemble_mean_std_max_min(`*ens: Dataset*`)` → Dataset

Calculate ensemble statistics between a results from an ensemble of climate simulations.

Returns an xarray Dataset containing ensemble mean, standard-deviation, minimum and maximum for input climate simulations.

Parameters

ens (*xr.Dataset*) – Ensemble dataset (see xclim.ensembles.create_ensemble).

Returns

xr.Dataset – Dataset with data variables of ensemble statistics.

Examples

```
>>> from xclim.ensembles import create_ensemble, ensemble_mean_std_max_min
```

Create the ensemble dataset:

```
>>> ens = create_ensemble(temperature_datasets)
```

Calculate ensemble statistics:

```
>>> ens_mean_std = ensemble_mean_std_max_min(ens)
```

`xclim.ensembles._base.ensemble_percentiles(`*ens: xr.Dataset | xr.DataArray, values: Sequence[float] = [10, 50, 90], keep_chunk_size: bool | None = None, split: bool = True*`)` → xr.Dataset

Calculate ensemble statistics between a results from an ensemble of climate simulations.

Returns a Dataset containing ensemble percentiles for input climate simulations.

Parameters

- **ens** (*Union[xr.Dataset, xr.DataArray]*) – Ensemble dataset or dataarray (see xclim.ensembles.create_ensemble).

- **values** (*Tuple[int, int, int]*) – Percentile values to calculate. Default: (10, 50, 90).
- **keep_chunk_size** (*Optional[bool]*) – For ensembles using dask arrays, all chunks along the 'realization' axis are merged. If True, the dataset is rechunked along the dimension with the largest chunks, so that the chunks keep the same size (approx) If False, no shrinking is performed, resulting in much larger chunks If not defined, the function decides which is best
- **split** (*bool*) – Whether to split each percentile into a new variable of concatenate the ouput along a new "percentiles" dimension.

Returns

Union[xr.Dataset, xr.DataArray] – If split is True, same type as ens; dataset otherwise, containing data variable(s) of requested ensemble statistics

Examples

```
>>> from xclim.ensembles import create_ensemble, ensemble_percentiles
```

Create ensemble dataset:

```
>>> ens = create_ensemble(temperature_datasets)
```

Calculate default ensemble percentiles:

```
>>> ens_percs = ensemble_percentiles(ens)
```

Calculate non-default percentiles (25th and 75th)

```
>>> ens_percs = ensemble_percentiles(ens, values=(25, 50, 75))
```

If the original array has many small chunks, it might be more efficient to do:

```
>>> ens_percs = ensemble_percentiles(ens, keep_chunk_size=False)
```

xclim.ensembles._reduce module

Ensemble Reduction

Ensemble reduction is the process of selecting a subset of members from an ensemble in order to reduce the volume of computation needed while still covering a good portion of the simulated climate variability.

`xclim.ensembles._reduce._calc_rsq`(*z*, *method*, *make_graph*, *n_sim*, *random_state*, *sample_weights*)

Sub-function to kmeans_reduce_ensemble. Calculates r-square profile (r-square versus number of clusters.

`xclim.ensembles._reduce._get_nclust`(*method=None*, *n_sim=None*, *rsq=None*, *max_clusters=None*)

Sub-function to kmeans_reduce_ensemble. Determine number of clusters to create depending on various methods.

`xclim.ensembles._reduce.kkz_reduce_ensemble`(*data: DataArray*, *num_select: int*, ***, *dist_method: str = 'euclidean'*, *standardize: bool = True*, ***cdist_kwargs*) → list

Return a sample of ensemble members using KKZ selection.

The algorithm selects *num_select* ensemble members spanning the overall range of the ensemble. The selection is ordered, smaller groups are always subsets of larger ones for given criteria. The first selected member is the one nearest to the centroid of the ensemble, all subsequent members are selected in a way maximizing the phase-space coverage of the group. Algorithm taken from [CannonKKZ].

Parameters

- **data** (*xr.DataArray*) – Selection criteria data : 2-D xr.DataArray with dimensions 'realization' (N) and 'criteria' (P). These are the values used for clustering. Realizations represent the individual original ensemble members and criteria the variables/indicators used in the grouping algorithm.
- **num_select** (*int*) – The number of members to select.
- **dist_method** (*str*) – Any distance metric name accepted by *scipy.spatial.distance.cdist*.
- **standardize** (*bool*) – Whether to standardize the input before running the selection or not. Standardization consists in translation as to have a zero mean and scaling as to have a unit standard deviation.
- **cdist_kwargs** – All extra arguments are passed as-is to *scipy.spatial.distance.cdist*, see its docs for more information.

Returns

list – Selected model indices along the *realization* dimension.

References

`xclim.ensembles._reduce.kmeans_reduce_ensemble(`*data: xarray.DataArray, *, method: dict = None, make_graph: bool = True, max_clusters: int | None = None, variable_weights: np.ndarray | None = None, model_weights: np.ndarray | None = None, sample_weights: np.ndarray | None = None, random_state: int | np.random.RandomState | None = None*`)` → tuple[list, np.ndarray, dict]

Return a sample of ensemble members using k-means clustering.

The algorithm attempts to reduce the total number of ensemble members while maintaining adequate coverage of the ensemble uncertainty in an N-dimensional data space. K-Means clustering is carried out on the input selection criteria data-array in order to group individual ensemble members into a reduced number of similar groups. Subsequently, a single representative simulation is retained from each group.

Parameters

- **data** (*xr.DataArray*) – Selecton criteria data : 2-D xr.DataArray with dimensions 'realization' (N) and 'criteria' (P). These are the values used for clustering. Realizations represent the individual original ensemble members and criteria the variables/indicators used in the grouping algorithm.
- **method** (*dict*) – Dictionary defining selection method and associated value when required. See Notes.

- **max_clusters** (*Optional[int]*) – Maximum number of members to include in the output ensemble selection. When using 'rsq_optimize' or 'rsq_cutoff' methods, limit the final selection to a maximum number even if method results indicate a higher value. Defaults to N.
- **variable_weights** (*Optional[np.ndarray]*) – An array of size P. This weighting can be used to influence of weight of the climate indices (criteria dimension) on the clustering itself.
- **model_weights** (*Optional[np.ndarray]*) – An array of size N. This weighting can be used to influence which realization is selected from within each cluster. This parameter has no influence on the clustering itself.
- **sample_weights** (*Optional[np.ndarray]*) – An array of size N. sklearn.cluster.KMeans() sample_weights parameter. This weighting can be used to influence of weight of simulations on the clustering itself. See: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
- **random_state** (*Optional[Union[int, np.random.RandomState]]*) – sklearn.cluster.KMeans() random_state parameter. Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. See: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
- **make_graph** (*bool*) – output a dictionary of input for displays a plot of R^2 vs. the number of clusters. Defaults to True if matplotlib is installed in runtime environment.

Notes

Parameters for method in call must follow these conventions:

rsq_optimize

Calculate coefficient of variation (R^2) of cluster results for n = 1 to N clusters and determine an optimal number of clusters that balances cost / benefit tradeoffs. This is the default setting. See supporting information S2 text in https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0152495

method={'rsq_optimize':None}

rsq_cutoff

Calculate Coefficient of variation (R^2) of cluster results for n = 1 to N clusters and determine the minimum numbers of clusters needed for $R^2 >$ val.

val : float between 0 and 1. R^2 value that must be exceeded by clustering results.

method={'rsq_cutoff': val}

n_clusters

Create a user determined number of clusters.

val : integer between 1 and N

method={'n_clusters': val}

Returns

- *list* – Selected model indexes (positions)
- *np.ndarray* – KMeans clustering results

- *dict* – Dictionary of input data for creating R^2 profile plot. 'None' when make_graph=False

References

Casajus et al. 2016. https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0152495

Examples

```
>>> import xclim
>>> from xclim.ensembles import create_ensemble, kmeans_reduce_ensemble
>>> from xclim.indices import hot_spell_frequency
```

Start with ensemble datasets for temperature:

```
>>> ensTas = create_ensemble(temperature_datasets)
```

Calculate selection criteria – Use annual climate change fields between 2071-2100 and 1981-2010 normals. First, average annual temperature:

```
>>> tg = xclim.atmos.tg_mean(tas=ensTas.tas)
>>> his_tg = tg.sel(time=slice("1990", "2019")).mean(dim="time")
>>> fut_tg = tg.sel(time=slice("2020", "2050")).mean(dim="time")
>>> dtg = fut_tg - his_tg
```

Then, Hotspell frequency as second indicator:

```
>>> hs = hot_spell_frequency(tasmax=ensTas.tas, window=2, thresh_tasmax="10 degC")
>>> his_hs = hs.sel(time=slice("1990", "2019")).mean(dim="time")
>>> fut_hs = hs.sel(time=slice("2020", "2050")).mean(dim="time")
>>> dhs = fut_hs - his_hs
```

Create a selection criteria xr.DataArray:

```
>>> from xarray import concat
>>> crit = concat((dtg, dhs), dim="criteria")
```

Finally, create clusters and select realization ids of reduced ensemble:

```
>>> ids, cluster, fig_data = kmeans_reduce_ensemble(
...     data=crit, method={"rsq_cutoff": 0.9}, random_state=42, make_graph=False
... )
>>> ids, cluster, fig_data = kmeans_reduce_ensemble(
...     data=crit, method={"rsq_optimize": None}, random_state=42, make_graph=True
... )
```

`xclim.ensembles._reduce.plot_rsqprofile(`*fig_data*`)`

Create an R^2 profile plot using kmeans_reduce_ensemble output.

The R^2 plot allows evaluation of the proportion of total uncertainty in the original ensemble that is provided by the reduced selected.

Examples

```
>>> from xclim.ensembles import kmeans_reduce_ensemble, plot_rsqprofile
>>> is_matplotlib_installed()
>>> crit = xr.open_dataset(path_to_ensemble_file).data
>>> ids, cluster, fig_data = kmeans_reduce_ensemble(
...     data=crit, method={"rsq_cutoff": 0.9}, random_state=42, make_graph=True
... )
>>> plot_rsqprofile(fig_data)
```

xclim.ensembles._robustness module

Ensemble Robustness metrics.

Robustness metrics are used to estimate the confidence of the climate change signal of an ensemble. This submodule is inspired by and tries to follow the guidelines of the IPCC, more specifically the 12th chapter of the Working Group 1's contribution to the AR5 [AR5WG1C12] (see box 12.1).

References

`xclim.ensembles._robustness.change_significance`(*fut: xr.DataArray | xr.Dataset*, *ref: xr.DataArray | xr.Dataset = None*, *test: str = 'ttest'*, ***kwargs*) → tuple[xr.DataArray | xr.Dataset, xr.DataArray | xr.Dataset]

Robustness statistics qualifying how the members of an ensemble agree on the existence of change and on its sign.

Parameters

- **fut** (*Union[xr.DataArray, xr.Dataset]*) – Future period values along 'realization' and 'time' (..., nr, nt1) or if *ref* is None, Delta values along *realization* (..., nr).
- **ref** (*Union[xr.DataArray, xr.Dataset], optional*) – Reference period values along realization' and 'time' (..., nt2, nr). The size of the 'time' axis does not need to match the one of *fut*. But their 'realization' axes must be identical. If *None* (default), values of *fut* are assumed to be deltas instead of a distribution across the future period. *fut* and *ref* must be of the same type (Dataset or DataArray). If they are Dataset, they must have the same variables (name and coords).
- **test** (*{'ttest', 'welch-ttest', 'threshold', None}*) – Name of the statistical test used to determine if there was significant change. See notes.
- **kwargs** – Other arguments specific to the statistical test.

  For 'ttest' and 'welch-ttest':

  p_change
  [float (default][0.05)] p-value threshold for rejecting the hypothesis of no significant change.

  For 'threshold': (Only one of those must be given.)

  abs_thresh
  [float (no default)] Threshold for the (absolute) change to be considered significative.

rel_thresh
[float (no default, in [0, 1])] Threshold for the relative change (in reference to ref) to be significative. Only valid if *ref* is given.

Returns

- *change_frac* – The fraction of members that show significant change [0, 1]. Passing *test=None* yields change_frac = 1 everywhere. Same type as *fut*.
- *pos_frac* – The fraction of members showing significant change that show a positive change]0, 1]. Null values are returned where no members show significant change.

  The table below shows the coefficient needed to retrieve the number of members that have the indicated characteristics, by multiplying it to the total number of members (*fut.realization.size*).

	Significant change	Non-significant change
Any direction	change_frac	1 - change_frac
Positive change	pos_frac * change_frac	N.A.
Negative change	(1 - pos_frac) * change_frac	

Notes

Available statistical tests are :

'ttest' :
Single sample T-test. Same test as used by [tebaldi2011]. The future values are compared against the reference mean (over 'time'). Change is qualified as 'significant' when the test's p-value is below the user-provided *p_change* value.

'welch-ttest' :
Two-sided T-test, without assuming equal population variance. Same significance criterion as 'ttest'.

'threshold' :
Change is considered significative if the absolute delta exceeds a given threshold (absolute or relative).

None :
Significant change is not tested and, thus, members showing no change are included in the *sign_frac* output.

References

Example

This example computes the mean temperature in an ensemble and compares two time periods, qualifying significant change through a single sample T-test.

```
>>> from xclim import ensembles
>>> ens = ensembles.create_ensemble(temperature_datasets)
>>> tgmean = xclim.atmos.tg_mean(tas=ens.tas, freq="YS")
>>> fut = tgmean.sel(time=slice("2020", "2050"))
>>> ref = tgmean.sel(time=slice("1990", "2020"))
>>> chng_f, pos_f = ensembles.change_significance(fut, ref, test="ttest")
```

If the deltas were already computed beforehand, the 'threshold' test can still be used, here with a 2 K threshold.

```
>>> delta = fut.mean("time") - ref.mean("time")
>>> chng_f, pos_f = ensembles.change_significance(
...     delta, test="threshold", abs_thresh=2
... )
```

`xclim.ensembles._robustness.robustness_coefficient(`*fut: xr.DataArray | xr.Dataset*, *ref: xr.DataArray | xr.Dataset*`)` → xr.DataArray | xr.Dataset

Robustness coefficient quantifying the robustness of a climate change signal in an ensemble.

Taken from Knutti and Sedlacek (2013).

The robustness metric is defined as R = 1 A1 / A2 , where A1 is defined as the integral of the squared area between two cumulative density functions characterizing the individual model projections and the multi-model mean projection and A2 is the integral of the squared area between two cumulative density functions characterizing the multi-model mean projection and the historical climate. (Description taken from [knutti2013])

A value of R equal to one implies perfect model agreement. Higher model spread or smaller signal decreases the value of R.

Parameters

- **fut** (*Union[xr.DataArray, xr.Dataset]*) – Future ensemble values along 'realization' and 'time' (nr, nt). Can be a dataset, in which case the coeffcient is computed on each variables.
- **ref** (*Union[xr.DataArray, xr.Dataset]*) – Reference period values along 'time' (nt). Same type as *fut*.

Returns

R – The robustness coeffcient,]-inf, 1], float. Same type as *fut* or *ref*.

References

xclim.indicators package

Indicators module

Indicators are the main tool xclim provides to compute climate indices. In contrast to the function defined in *xclim.indices*, Indicators add a layer of health checks and metadata handling. Indicator objects are split into realms : atmos, land and seaIce.

Virtual modules are also inserted here. A normal installation of xclim comes with three virtual modules:

- `xclim.indicators.cf`, Indicators defined in *cf-index-meta*.
- `xclim.indicators.icclim`, Indicators defined by ECAD, as found in python package Icclim.
- `xclim.indicators.anuclim`, Indicators of the Australian National University's Fenner School of Environment and Society.

Subpackages

xclim.indicators.atmos package

Atmospheric indicators

While the *indices* module stores the computing functions, this module defines Indicator classes and instances that include a number of functionalities, such as input validation, unit conversion, output meta-data handling, and missing value masking.

The concept followed here is to define Indicator subclasses for each input variable, then create instances for each indicator.

Submodules

xclim.indicators.atmos._conversion module

Atmospheric conversion definitions.

`xclim.indicators.atmos._conversion.corn_heat_units(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '4.44 degC', thresh_tasmax: str = '10 degC', ds: Dataset = None*`)` → DataArray

Corn heat units. (realm: atmos)

Temperature-based index used to estimate the development of corn crops. Formula adapted from [BootsmaTremblay&Filion1999]_.

This indicator will check for missing values according to the method "skip". Based on indice `corn_heat_units()`.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The minimum temperature threshold needed for corn growth. Default : 4.44 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The maximum temperature threshold needed for corn growth. Default : 10 degC. [Required units : [temperature]]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

chu (*DataArray*) – Corn heat units (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax}). description: Temperature-based index used to estimate the development of corn crops. Corn growth occurs when the minimum and maximum daily temperature both exceeds specific thresholds : Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax}.

Notes

Formula used in calculating the Corn Heat Units for the Agroclimatic Atlas of Quebec [Audet&al2012]_.

The thresholds of 4.44°C for minimum temperatures and 10°C for maximum temperatures were selected following the assumption that no growth occurs below these values.

Let TX_i and TN_i be the daily maximum and minimum temperature at day i. Then the daily corn heat unit is:

$$CHU_i = \frac{YX_i + YN_i}{2}$$

with

$$\begin{aligned} YX_i &= 3.33(TX_i - 10) - 0.084(TX_i - 10)^2, && \text{if } TX_i > 10C \\ YN_i &= 1.8(TN_i - 4.44), && \text{if } TN_i > 4.44C \end{aligned}$$

where YX_i and YN_i is 0 when $TX_i \leq 10C$ and $TN_i \leq 4.44C$, respectively.

References

`xclim.indicators.atmos._conversion.heat_index(`*tasmax: Union[DataArray, str] = 'tasmax', hurs: Union[DataArray, str] = 'hurs', *, ds: Dataset = None*`)` → DataArray

Daily heat index. (realm: atmos)

Perceived temperature after relative humidity is taken into account ([Blazejczyk2012]). The index is only valid for temperatures above 20°C.

Based on indice *heat_index()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **hurs** (*str or DataArray*) – Relative humidity. Default : *ds.hurs*. [Required units : []]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

heat_index (*DataArray*) – heat index (air_temperature) [C] description: Perceived temperature after relative humidity is taken into account.

Notes

While both the humidex and the heat index are calculated using dew point, the humidex uses a dew point of 7 °C (45 °F) as a base, whereas the heat index uses a dew point base of 14 °C (57 °F). Further, the heat index uses heat balance equations which account for many variables other than vapor pressure, which is used exclusively in the humidex calculation.

References

`xclim.indicators.atmos._conversion.humidex(`*tas: Union[DataArray, str] = 'tas', tdps: Optional[Union[DataArray, str]] = None, hurs: Optional[Union[DataArray, str]] = None, *, ds: Dataset = None*`)` → DataArray

Humidex index. (realm: atmos)

The humidex indicates how hot the air feels to an average person, accounting for the effect of humidity. It can be loosely interpreted as the equivalent perceived temperature when the air is dry.

Based on indice `humidex()`.

Parameters

- **tas** (*str or DataArray*) – Air temperature. Default : *ds.tas*. [Required units : [temperature]]
- **tdps** (*str or DataArray, optional*) – Dewpoint temperature. [Required units : [temperature]]
- **hurs** (*str or DataArray, optional*) – Relative humidity. [Required units : []]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

humidex (*DataArray*) – humidex index (air_temperature) [C] description: Humidex index describing the temperature felt by the average person in response to relative humidity.

Notes

The humidex is usually computed using hourly observations of dry bulb and dewpoint temperatures. It is computed using the formula based on [masterton79]:

$$T + \frac{5}{9}\left[e - 10\right]$$

where T is the dry bulb air temperature (°C). The term e can be computed from the dewpoint temperature $T_{dewpoint}$ in °K:

$$e = 6.112 \times \exp(5417.7530\left(\frac{1}{273.16} - \frac{1}{T_{\text{dewpoint}}}\right)$$

where the constant 5417.753 reflects the molecular weight of water, latent heat of vaporization, and the universal gas constant ([mekis15]). Alternatively, the term e can also be computed from the relative humidity h expressed in percent using [sirangelo20]:

$$e = \frac{h}{100} \times 6.112 * 10^{7.5 T/(T + 237.7)}.$$

The humidex *comfort scale* ([eccc]) can be interpreted as follows:

- 20 to 29 : no discomfort;
- 30 to 39 : some discomfort;
- 40 to 45 : great discomfort, avoid exertion;
- 46 and over : dangerous, possible heat stroke;

Please note that while both the humidex and the heat index are calculated using dew point, the humidex uses a dew point of 7 °C (45 °F) as a base, whereas the heat index uses a dew point base of 14 °C (57 °F). Further, the heat index uses heat balance equations which account for many variables other than vapor pressure, which is used exclusively in the humidex calculation.

References

`xclim.indicators.atmos._conversion.mean_radiant_temperature(`*rsds: Union[DataArray, str] = 'rsds', rsus: Union[DataArray, str] = 'rsus', rlds: Union[DataArray, str] = 'rlds', rlus: Union[DataArray, str] = 'rlus', *, stat: str = 'average', ds: Dataset = None*`)` → DataArray

Mean radiant temperature. (realm: atmos)

The mean radiant temperature is the incidence of radiation on the body from all directions. WARNING: There are some issues in the calculation of mrt in polar regions.

Based on indice *mean_radiant_temperature()*.

Parameters

- **rsds** (*str or DataArray*) – Surface Downwelling Shortwave Radiation Default : *ds.rsds*. [Required units : [radiation]]
- **rsus** (*str or DataArray*) – Surface Upwelling Shortwave Radiation Default : *ds.rsus*. [Required units : [radiation]]
- **rlds** (*str or DataArray*) – Surface Downwelling Longwave Radiation Default : *ds.rlds*. [Required units : [radiation]]
- **rlus** (*str or DataArray*) – Surface Upwelling Longwave Radiation Default : *ds.rlus*. [Required units : [radiation]]
- **stat** (*{'average', 'sunlit', 'instant'}*) – Which statistic to apply. If "average", the average of the cosine of the solar zenith angle is calculated. If "instant", the instantaneous cosine of the solar zenith angle is calculated. If "sunlit", the cosine of the solar zenith angle is calculated during the sunlit period of each interval. If "instant", the instantaneous cosine of the solar zenith angle is calculated. This is necessary if mrt is not None. Default : average.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

mrt (*DataArray*) – Mean radiant temperature [K] description: The incidence of radiation on the body from all directions.

Notes

This code was inspired by the *thermofeel* package.

References

Di Napoli, C., Hogan, R.J. & Pappenberger, F. Mean radiant temperature from global-scale numerical weather prediction models. Int J Biometeorol 64, 1233–1245 (2020). https://doi.org/10.1007/s00484-020-01900-5 Brimicombe , C., Di Napoli, C., Quintino, T., Pappenberger, F., Cornforth, R. and Cloke, H., 2021 thermofeel: a python thermal comfort indices library, https://doi.org/10.21957/mp6v-fd16

`xclim.indicators.atmos._conversion.potential_evapotranspiration(`*tasmin: Optional[Union[DataArray, str]] = None, tasmax: Optional[Union[DataArray, str]] = None, tas: Optional[Union[DataArray, str]] = None, lat: Optional[Union[DataArray, str]] = None, *, method: str = 'BR65', peta: float | None = 0.00516409319477, petb: float | None = 0.0874972822289, ds: Dataset = None*`)` → DataArray

Potential evapotranspiration. (realm: atmos)

The potential for water evaporation from soil and transpiration by plants if the water supply is sufficient, according to a given method.

Based on indice `potential_evapotranspiration()`.

Parameters

- **tasmin** (*str or DataArray, optional*) – Minimum daily temperature. [Required units : [temperature]]
- **tasmax** (*str or DataArray, optional*) – Maximum daily temperature. [Required units : [temperature]]
- **tas** (*str or DataArray, optional*) – Mean daily temperature. [Required units : [temperature]]
- **lat** (*str or DataArray, optional*) – Latitude. If not given, it is sought on tasmin or tas with cf-xarray. [Required units : []]
- **method** (*{'hargreaves85', 'baierrobertson65', 'HG85', 'MB05', 'mcguinnessbordne05', 'thornthwaite48', 'BR65', 'TW48'}*) – Which method to use, see notes. Default : BR65.
- **peta** (*number*) – Used only with method MB05 as a for calculation of PET, see Notes section. Default value resulted from calibration of PET over the UK. Default : 0.00516409319477.
- **petb** (*number*) – Used only with method MB05 as b for calculation of PET, see Notes section. Default value resulted from calibration of PET over the UK. Default : 0.0874972822289.

- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

evspsblpot (*DataArray*) – Potential evapotranspiration (water_potential_evapotranspiration_flux) [kg m-2 s-1] description: The potential for water evaporation from soil and transpiration by plants if the water supply is sufficient, with the method {method}.

Notes

Available methods are:

- "baierrobertson65" or "BR65", based on [BaierRobertson1965]. Requires tasmin and tasmax, daily [D] freq.
- "hargreaves85" or "HG85", based on [Hargreaves1985]. Requires tasmin and tasmax, daily [D] freq. (optional: tas can be given in addition of tasmin and tasmax).
- "mcguinnessbordne05" or "MB05", based on [Tanguy2018]. Requires tas, daily [D] freq, with latitudes 'lat'.
- "thornthwaite48" or "TW48", based on [Thornthwaite1948]. Requires tasmin and tasmax, monthly [MS] or daily [D] freq. (optional: tas can be given instead of tasmin and tasmax).

The McGuinness-Bordne [McGuinness1972] equation is:

$$PET[mmday^{-1}] = a * \frac{S_0}{\lambda}T_a + b * S_0\lambda$$

where a and b are empirical parameters; S_0 is the extraterrestrial radiation [MJ m-2 day-1], assuming a solar constant of 1367 W m-2;
$lambda$ is the latent heat of vaporisation [MJ kg-1] and T_a is the air temperature [°C]. The equation was originally derived for the USA, with $a = 0.0147$ and $b = 0.07353$. The default parameters used here are calibrated for the UK, using the method described in [Tanguy2018].

Methods "BR65", "HG85" and "MB05" use an approximation of the extraterrestrial radiation. See `extraterrestrial_solar_radiation()`.

References

`xclim.indicators.atmos._conversion.rain_approximation(`*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', method: str = 'binary', ds: Dataset = None*`)` → DataArray

Rainfall approximation from total precipitation and temperature. (realm: atmos)

Liquid precipitation estimated from precipitation and temperature according to a given method. This is a convenience method based on *`snowfall_approximation()`*, see the latter for details.

Based on indice *`rain_approximation()`*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean, maximum, or minimum daily temperature. Default : *ds.tas*. [Required units : [temperature]]

- **thresh** (*quantity (string with units)*) – Threshold temperature, used by method "binary". Default : 0 degC. [Required units : [temperature]]
- **method** (*{'brown', 'auer', 'binary'}*) – Which method to use when approximating snowfall from total precipitation. See notes. Default : binary.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

prlp (*DataArray*) – Liquid precipitation (precipitation_flux) [kg m-2 s-1] description: Liquid precipitation estimated from total precipitation and temperature with method {method} and threshold temperature {thresh}.

Notes

This method computes the snowfall approximation and subtracts it from the total precipitation to estimate the liquid rain precipitation.

`xclim.indicators.atmos._conversion.relative_humidity`(*tas: Union[DataArray, str] = 'tas', huss: Union[DataArray, str] = 'huss', ps: Union[DataArray, str] = 'ps', *, ice_thresh: str = None, method: str = 'sonntag90', ds: Dataset = None*) → DataArray

Relative humidity from temperature, pressure and specific humidity. (realm: atmos)

Compute relative humidity from temperature and either dewpoint temperature or specific humidity and pressure through the saturation vapor pressure.

Based on indice `relative_humidity()`. With injected parameters: tdps=None, invalid_values=mask.

Parameters

- **tas** (*str or DataArray*) – Temperature array Default : *ds.tas*. [Required units : [temperature]]
- **huss** (*str or DataArray*) – Specific humidity. Default : *ds.huss*. [Required units : []]
- **ps** (*str or DataArray*) – Air Pressure. Default : *ds.ps*. [Required units : [pressure]]
- **ice_thresh** (*quantity (string with units)*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water. Does nothing if 'method' is "bohren98". Default : None. [Required units : [temperature]]
- **method** (*{'goffgratch46', 'wmo08', 'sonntag90', 'bohren98', 'tetens30'}*) – Which method to use, see notes of this function and of *saturation_vapor_pressure*. Default : sonntag90.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

hurs (*DataArray*) – Relative Humidity (relative_humidity) [%] description: <Dynamically generated string>

Notes

In the following, let T, T_d, q and p be the temperature, the dew point temperature, the specific humidity and the air pressure.

For the "bohren98" method : This method does not use the saturation vapor pressure directly, but rather uses an approximation of the ratio of $\frac{e_{sat}(T_d)}{e_{sat}(T)}$. With L the enthalpy of vaporization of water and R_w the gas constant for water vapor, the relative humidity is computed as:

$$RH = e^{\frac{-L(T-T_d)}{R_w T T_d}}$$

From [BohrenAlbrecht1998], formula taken from [Lawrence2005]. $L = 2.5 \times 10^{-6}$ J kg-1, exact for $T = 273.15$ K, is used.

Other methods: With w, w_{sat}, e_{sat} the mixing ratio, the saturation mixing ratio and the saturation vapor pressure. If the dewpoint temperature is given, relative humidity is computed as:

$$RH = 100\frac{e_{sat}(T_d)}{e_{sat}(T)}$$

Otherwise, the specific humidity and the air pressure must be given so relative humidity can be computed as:

$$RH = 100\frac{w}{w_{sat}}w = \frac{q}{1-q}w_{sat} = 0.622\frac{e_{sat}}{P - e_{sat}}$$

The methods differ by how e_{sat} is computed. See the doc of `xclim.core.utils.saturation_vapor_pressure()`.

References

`xclim.indicators.atmos._conversion.relative_humidity_from_dewpoint(`*tas: Union[DataArray, str] = 'tas', tdps: Union[DataArray, str] = 'tdps', *, ice_thresh: str = None, method: str = 'sonntag90', ds: Dataset = None*`)` → DataArray

Relative humidity from temperature and dewpoint temperature. (realm: atmos)

Compute relative humidity from temperature and either dewpoint temperature or specific humidity and pressure through the saturation vapor pressure.

Based on indice *`relative_humidity()`*. With injected parameters: huss=None, ps=None, invalid_values=mask.

Parameters

- **tas** (*str or DataArray*) – Temperature array Default : *ds.tas.* [Required units : [temperature]]
- **tdps** (*str or DataArray*) – Dewpoint temperature, if specified, overrides huss and ps. Default : *ds.tdps.* [Required units : [temperature]]
- **ice_thresh** (*quantity (string with units)*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water. Does nothing if 'method' is "bohren98". Default : None. [Required units : [temperature]]

- **method** (*{'goffgratch46', 'wmo08', 'sonntag90', 'bohren98', 'tetens30'}*) – Which method to use, see notes of this function and of *saturation_vapor_pressure*. Default : sonntag90.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
hurs (*DataArray*) – Relative Humidity (relative_humidity) [%] description: <Dynamically generated string>

Notes

In the following, let T, T_d, q and p be the temperature, the dew point temperature, the specific humidity and the air pressure.

For the "bohren98" method : This method does not use the saturation vapor pressure directly, but rather uses an approximation of the ratio of $\frac{e_{sat}(T_d)}{e_{sat}(T)}$. With L the enthalpy of vaporization of water and R_w the gas constant for water vapor, the relative humidity is computed as:

$$RH = e^{\frac{-L(T-T_d)}{R_w T T_d}}$$

From [BohrenAlbrecht1998], formula taken from [Lawrence2005]. $L = 2.5 \times 10^{-6}$ J kg-1, exact for $T = 273.15$ K, is used.

Other methods: With w, w_{sat}, e_{sat} the mixing ratio, the saturation mixing ratio and the saturation vapor pressure. If the dewpoint temperature is given, relative humidity is computed as:

$$RH = 100\frac{e_{sat}(T_d)}{e_{sat}(T)}$$

Otherwise, the specific humidity and the air pressure must be given so relative humidity can be computed as:

$$RH = 100\frac{w}{w_{sat}} w = \frac{q}{1-q} w_{sat} = 0.622\frac{e_{sat}}{P - e_{sat}}$$

The methods differ by how e_{sat} is computed. See the doc of `xclim.core.utils.saturation_vapor_pressure()`.

References

`xclim.indicators.atmos._conversion.saturation_vapor_pressure(`*tas: Union[DataArray, str] = 'tas', *, ice_thresh: str = None, method: str = 'sonntag90', ds: Dataset = None*`)` → DataArray

Saturation vapor pressure from temperature. (realm: atmos)

Based on indice *saturation_vapor_pressure()*.

Parameters

- **tas** (*str or DataArray*) – Temperature array. Default : *ds.tas*. [Required units : [temperature]]
- **ice_thresh** (*quantity (string with units)*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water. Default : None. [Required units : [temperature]]

- **method** (*{'goffgratch46', 'wmo08', 'sonntag90', 'its90', 'tetens30'}*) – Which method to use, see notes. Default : sonntag90.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

e_sat (*DataArray*) – Saturation vapor pressure [Pa] description: <Dynamically generated string>

Notes

In all cases implemented here $log(e_{sat})$ is an empirically fitted function (usually a polynomial) where coefficients can be different when ice is taken as reference instead of water. Available methods are:

- "goffgratch46" or "GG46", based on [goffgratch46], values and equation taken from [voemel].
- "sonntag90" or "SO90", taken from [sonntag90].
- "tetens30" or "TE30", based on [tetens30], values and equation taken from [voemel].
- "wmo08" or "WMO08", taken from [wmo08].
- "its90" or "ITS90", taken from [its90].

References

`xclim.indicators.atmos._conversion.snowfall_approximation(`*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', method: str = 'binary', ds: Dataset = None*`)` → DataArray

Snowfall approximation from total precipitation and temperature. (realm: atmos)

Solid precipitation estimated from precipitation and temperature according to a given method.

Based on indice *snowfall_approximation()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean, maximum, or minimum daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature, used by method "binary". Default : 0 degC. [Required units : [temperature]]
- **method** (*{'brown', 'auer', 'binary'}*) – Which method to use when approximating snowfall from total precipitation. See notes. Default : binary.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

prsn (*DataArray*) – Solid precipitation (solid_precipitation_flux) [kg m-2 s-1] description: Solid precipitation estimated from total precipitation and temperature with method {method} and threshold temperature {thresh}.

Notes

The following methods are available to approximate snowfall and are drawn from the Canadian Land Surface Scheme (CLASS, [Verseghy09]).

- `'binary'` : When the temperature is under the freezing threshold, precipitation is assumed to be solid. The method is agnostic to the type of temperature used (mean, maximum or minimum).
- `'brown'` : The phase between the freezing threshold goes from solid to liquid linearly over a range of 2°C over the freezing point.
- `'auer'` : The phase between the freezing threshold goes from solid to liquid as a degree six polynomial over a range of 6°C over the freezing point.

References

https://gitlab.com/cccma/classic/-/blob/master/src/atmosphericVarsCalc.f90

`xclim.indicators.atmos._conversion.specific_humidity(`*tas: Union[DataArray, str] = 'tas', hurs: Union[DataArray, str] = 'hurs', ps: Union[DataArray, str] = 'ps', *, ice_thresh: str = None, method: str = 'sonntag90', ds: Dataset = None*`)` → DataArray

Specific humidity from temperature, relative humidity and pressure. (realm: atmos)

Specific humidity is the ratio between the mass of water vapour and the mass of moist air [WMO08].

Based on indice *`specific_humidity()`*. With injected parameters: invalid_values=mask.

Parameters

- **tas** (*str or DataArray*) – Temperature array Default : *ds.tas*. [Required units : [temperature]]
- **hurs** (*str or DataArray*) – Relative Humidity. Default : *ds.hurs*. [Required units : []]
- **ps** (*str or DataArray*) – Air Pressure. Default : *ds.ps*. [Required units : [pressure]]
- **ice_thresh** (*quantity (string with units)*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water. Default : None. [Required units : [temperature]]
- **method** (*{'wmo08', 'goffgratch46', 'tetens30', 'sonntag90'}*) – Which method to use, see notes of this function and of *saturation_vapor_pressure*. Default : sonntag90.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

huss (*DataArray*) – Specific Humidity (specific_humidity) description: <Dynamically generated string>

Notes

In the following, let T, $hurs$ (in %) and p be the temperature, the relative humidity and the air pressure. With w, w_{sat}, e_{sat} the mixing ratio, the saturation mixing ratio and the saturation vapor pressure, specific humidity q is computed as:

$$w_{sat} = 0.622\frac{e_{sat}}{P - e_{sat}}w = w_{sat} * hurs/100q = w/(1 + w)$$

The methods differ by how e_{sat} is computed. See the doc of *xclim.core.utils.saturation_vapor_pressure*.

If *invalid_values* is not *None*, the saturation specific humidity q_{sat} is computed as:

$$q_{sat} = w_{sat}/(1 + w_{sat})$$

References

`xclim.indicators.atmos._conversion.specific_humidity_from_dewpoint(`*tdps: Union[DataArray, str] = 'tdps', ps: Union[DataArray, str] = 'ps', *, method: str = 'sonntag90', ds: Dataset = None*) → DataArray

Specific humidity from dewpoint temperature and air pressure. (realm: atmos)

Specific humidity is the ratio between the mass of water vapour and the mass of moist air [WMO08].

Based on indice `specific_humidity_from_dewpoint()`.

Parameters

- **tdps** (*str or DataArray*) – Dewpoint temperature array. Default : *ds.tdps*. [Required units : [temperature]]
- **ps** (*str or DataArray*) – Air pressure array. Default : *ds.ps*. [Required units : [pressure]]
- **method** (*{'wmo08', 'goffgratch46', 'tetens30', 'sonntag90'}*) – Method to compute the saturation vapor pressure. Default : sonntag90.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

huss_fromdewpoint (*DataArray*) – Specific Humidity (specific_humidity) description: Computed from dewpoint temperature and pressure through the saturation vapor pressure, which was calculated according to the {method} method.

Notes

If e is the water vapor pressure, and p the total air pressure, then specific humidity is given by

$$q = m_w e/(m_a(p - e) + m_w e)$$

where m_w and m_a are the molecular weights of water and dry air respectively. This formula is often written with $= m_w/m_a$, which simplifies to $q = e/(p - e(1-))$.

References

`xclim.indicators.atmos._conversion.tg`(*tasmin: Union[DataArray, str] = 'tasmin'*, *tasmax: Union[DataArray, str] = 'tasmax'*, ***, *ds: Dataset = None*) → DataArray

Average temperature from minimum and maximum temperatures. (realm: atmos)

We assume a symmetrical distribution for the temperature and retrieve the average value as Tg = (Tx + Tn) / 2

Based on indice *tas()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum (daily) temperature Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum (daily) temperature Default : *ds.tasmax*. [Required units : [temperature]]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

tg (*DataArray*) – Daily mean temperature (air_temperature) [K] cell_methods: time: mean within days description: Estimated mean temperature from maximum and minimum temperatures

`xclim.indicators.atmos._conversion.universal_thermal_climate_index`(*tas: Union[DataArray, str] = 'tas'*, *hurs: Union[DataArray, str] = 'hurs'*, *sfcWind: Union[DataArray, str] = 'sfcWind'*, *mrt: Optional[Union[DataArray, str]] = None*, *rsds: Optional[Union[DataArray, str]] = None*, *rsus: Optional[Union[DataArray, str]] = None*, *rlds: Optional[Union[DataArray, str]] = None*, *rlus: Optional[Union[DataArray, str]] = None*, ***, *stat: str = 'average'*, *mask_invalid: bool = True*, *ds: Dataset = None*) → DataArray

Universal thermal climate index. (realm: atmos)

The UTCI is the equivalent temperature for the environment derived from a reference environment and is used to evaluate heat stress in outdoor spaces.

Based on indice *universal_thermal_climate_index()*.

Parameters

- **tas** (*str or DataArray*) – Mean temperature Default : *ds.tas*. [Required units : [temperature]]
- **hurs** (*str or DataArray*) – Relative Humidity Default : *ds.hurs*. [Required units : []]

- **sfcWind** (*str or DataArray*) – Wind velocity Default : *ds.sfcWind*. [Required units : [speed]]
- **mrt** (*str or DataArray, optional*) – Mean radiant temperature [Required units : [temperature]]
- **rsds** (*str or DataArray, optional*) – Surface Downwelling Shortwave Radiation This is necessary if mrt is not None. [Required units : [radiation]]
- **rsus** (*str or DataArray, optional*) – Surface Upwelling Shortwave Radiation This is necessary if mrt is not None. [Required units : [radiation]]
- **rlds** (*str or DataArray, optional*) – Surface Downwelling Longwave Radiation This is necessary if mrt is not None. [Required units : [radiation]]
- **rlus** (*str or DataArray, optional*) – Surface Upwelling Longwave Radiation This is necessary if mrt is not None. [Required units : [radiation]]
- **stat** (*{'average', 'sunlit', 'instant'}*) – Which statistic to apply. If "average", the average of the cosine of the solar zenith angle is calculated. If "instant", the instantaneous cosine of the solar zenith angle is calculated. If "sunlit", the cosine of the solar zenith angle is calculated during the sunlit period of each interval. If "instant", the instantaneous cosine of the solar zenith angle is calculated. This is necessary if mrt is not None. Default : average.
- **mask_invalid** (*boolean*) – If True (default), UTCI values are NaN where any of the inputs are outside their validity ranges : -50°C < tas < 50°C, -30°C < tas - mrt < 30°C and 0.5 m/s < sfcWind < 17.0 m/s. Default : True.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

utci (*DataArray*) – Universal Thermal Climate Index [K] description: UTCI is the equivalent temperature for the environment derived from a reference environment and is used to evaluate heat stress in outdoor spaces.

Notes

The calculation uses water vapor partial pressure, which is derived from relative humidity and saturation vapor pressure computed according to the ITS-90 equation.

This code was inspired by the *pythermalcomfort* and *thermofeel* packages.

References

Bröde, Peter (2009). Program for calculating UTCI Temperature (UTCI), version a 0.002, http://www.utci.org/public/UTCI%20Program%20Code/UTCI_a002.f90 Błażejczyk, K., Jendritzky, G., Bröde, P., Fiala, D., Havenith, G., Epstein, Y., Psikuta, A., & Kampmann, B. (2013). An introduction to the Universal Thermal Climate Index (UTCI). DOI:10.7163/GPOL.2013.1

`xclim.indicators.atmos._conversion.water_budget(`*pr: Union[DataArray, str] = 'pr', evspsblpot: Optional[Union[DataArray, str]] = None, tasmin: Optional[Union[DataArray, str]] = None, tasmax: Optional[Union[DataArray, str]] = None, tas: Optional[Union[DataArray, str]] = None, lat: Optional[Union[DataArray, str]] = None, *, ds: Dataset = None*`)` → DataArray

Precipitation minus potential evapotranspiration. (realm: atmos)

Precipitation minus potential evapotranspiration as a measure of an approximated surface water budget, where the potential evapotranspiration can be calculated with a given method.

Based on indice `water_budget()`. With injected parameters: method=dummy.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **evspsblpot** (*str or DataArray, optional*) – Potential evapotranspiration [Required units : [precipitation]]
- **tasmin** (*str or DataArray, optional*) – Minimum daily temperature. [Required units : [temperature]]
- **tasmax** (*str or DataArray, optional*) – Maximum daily temperature. [Required units : [temperature]]
- **tas** (*str or DataArray, optional*) – Mean daily temperature. [Required units : [temperature]]
- **lat** (*str or DataArray, optional*) – Latitude, needed if evspsblpot is not given. [Required units : []]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

water_budget (*DataArray*) – Water budget [kg m-2 s-1] description: Precipitation minus potential evapotranspiration as a measure of an approximated surface water budget.

Notes

Available methods are listed in the description of xclim.indicators.atmos.potential_evapotranspiration.

`xclim.indicators.atmos._conversion.water_budget_from_tas`(*pr: Union[DataArray, str] = 'pr', evspsblpot: Optional[Union[DataArray, str]] = None, tasmin: Optional[Union[DataArray, str]] = None, tasmax: Optional[Union[DataArray, str]] = None, tas: Optional[Union[DataArray, str]] = None, lat: Optional[Union[DataArray, str]] = None, *, method: str = 'BR65', ds: Dataset = None*) → DataArray

Precipitation minus potential evapotranspiration. (realm: atmos)

Precipitation minus potential evapotranspiration as a measure of an approximated surface water budget, where the potential evapotranspiration can be calculated with a given method.

Based on indice `water_budget()`.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]

- **evspsblpot** (*str or DataArray, optional*) – Potential evapotranspiration [Required units : [precipitation]]
- **tasmin** (*str or DataArray, optional*) – Minimum daily temperature. [Required units : [temperature]]
- **tasmax** (*str or DataArray, optional*) – Maximum daily temperature. [Required units : [temperature]]
- **tas** (*str or DataArray, optional*) – Mean daily temperature. [Required units : [temperature]]
- **lat** (*str or DataArray, optional*) – Latitude, needed if evspsblpot is not given. [Required units : []]
- **method** (*str*) – Method to use to calculate the potential evapotranspiration. Default : BR65.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
water_budget_from_tas (*DataArray*) – Water budget [kg m-2 s-1] description: Precipitation minus potential evapotranspiration as a measure of an approximated surface water budget, where the potential evapotranspiration is calculated with the method {method}.

Notes

Available methods are listed in the description of xclim.indicators.atmos.potential_evapotranspiration.

`xclim.indicators.atmos._conversion.wind_chill_index(`*tas: Union[DataArray, str] = 'tas', sfcWind: Union[DataArray, str] = 'sfcWind', *, method: str = 'CAN', ds: Dataset = None*) → DataArray

Wind chill index. (realm: atmos)

The Wind Chill Index is an estimation of how cold the weather feels to the average person. It is computed from the air temperature and the 10-m wind. As defined by the Environment and Climate Change Canada ([MVSZ2015]), two equations exist, the conventional one and one for slow winds (usually < 5 km/h), see Notes.

Based on indice *wind_chill_index()*. With injected parameters: mask_invalid=True.

Parameters

- **tas** (*str or DataArray*) – Surface air temperature. Default : *ds.tas*. [Required units : [temperature]]
- **sfcWind** (*str or DataArray*) – Surface wind speed (10 m). Default : *ds.sfcWind*. [Required units : [speed]]
- **method** (*{'US', 'CAN'}*) – If "CAN" (default), a "slow wind" equation is used where winds are slower than 5 km/h, see Notes. Default : CAN.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
wind_chill (*DataArray*) – Wind chill index [degC] description: <Dynamically generated string>

Notes

Following the calculations of Environment and Climate Change Canada, this function switches from the standardized index to another one for slow winds. The standard index is the same as used by the National Weather Service of the USA ([NWS]). Given a temperature at surface T (in °C) and 10-m wind speed V (in km/h), the Wind Chill Index W (dimensionless) is computed as:

$$W = 13.12 + 0.6125 * T - 11.37 * V^0.16 + 0.3965 * T * V^0.16$$

Under slow winds ($V < 5$ km/h), and using the canadian method, it becomes:

$$W = T + \frac{-1.59 + 0.1345 * T}{5} * V$$

Both equations are invalid for temperature over 0°C in the canadian method.

The american Wind Chill Temperature index (WCT), as defined by USA's National Weather Service, is computed when *method='US'*. In that case, the maximal valid temperature is 50°F (10 °C) and minimal wind speed is 3 mph (4.8 km/h).

References

`xclim.indicators.atmos._conversion.wind_speed_from_vector(`*uas: Union[DataArray, str] = 'uas', vas: Union[DataArray, str] = 'vas', *, calm_wind_thresh: str = '0.5 m/s', ds: Dataset = None*`)` → Tuple[DataArray, DataArray]

Wind speed and direction from the eastward and northward wind components. (realm: atmos)

Computes the magnitude and angle of the wind vector from its northward and eastward components, following the meteorological convention that sets calm wind to a direction of 0° and northerly wind to 360°.

Based on indice `uas_vas_2_sfcwind()`.

Parameters

- **uas** (*str or DataArray*) – Eastward wind velocity Default : *ds.uas*. [Required units : [speed]]
- **vas** (*str or DataArray*) – Northward wind velocity Default : *ds.vas*. [Required units : [speed]]
- **calm_wind_thresh** (*quantity (string with units)*) – The threshold under which winds are considered "calm" and for which the direction is set to 0. On the Beaufort scale, calm winds are defined as < 0.5 m/s. Default : 0.5 m/s. [Required units : [speed]]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

- **sfcWind** (*DataArray*) – Near-Surface Wind Speed (wind_speed) [m s-1] description: Wind speed computed as the magnitude of the (uas, vas) vector.
- **sfcWindfromdir** (*DataArray*) – Near-Surface Wind from Direction (wind_from_direction) [degree] description: Wind direction computed as the angle of the (uas, vas) vector. A direction of 0° is attributed to winds with a speed under {calm_wind_thresh}.

Notes

Winds with a velocity less than *calm_wind_thresh* are given a wind direction of 0°, while stronger northerly winds are set to 360°.

`xclim.indicators.atmos._conversion.wind_vector_from_speed(`*sfcWind: Union[DataArray, str] = 'sfcWind', sfcWindfromdir: Union[DataArray, str] = 'sfcWindfromdir', *, ds: Dataset = None*`)` → Tuple[DataArray, DataArray]

Eastward and northward wind components from the wind speed and direction. (realm: atmos)

Compute the eastward and northward wind components from the wind speed and direction.

Based on indice `sfcwind_2_uas_vas()`.

Parameters

- **sfcWind** (*str or DataArray*) – Wind velocity Default : *ds.sfcWind*. [Required units : [speed]]
- **sfcWindfromdir** (*str or DataArray*) – Direction from which the wind blows, following the meteorological convention where 360 stands for North. Default : *ds.sfcWindfromdir*. [Required units : []]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

- **uas** (*DataArray*) – Near-Surface Eastward Wind (eastward_wind) [m s-1] description: Eastward wind speed computed from its speed and direction of origin.
- **vas** (*DataArray*) – Near-Surface Northward Wind (northward_wind) [m s-1] description: Northward wind speed computed from its speed and direction of origin.

xclim.indicators.atmos._precip module

Precipitation indicator definitions.

`xclim.indicators.atmos._precip.cold_and_dry_days(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', tas_per: Union[DataArray, str] = 'tas_per', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Cold and dry days (realm: atmos)

Returns the total number of days where "Cold" and "Dry" conditions coincide.

This indicator will check for missing values according to the method "from_context". Based on indice `cold_and_dry_days()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature values Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]

- **tas_per** (*str or DataArray*) – First quartile of daily mean temperature computed by month. Default : *ds.tas_per*. [Required units : [temperature]]
- **pr_per** (*str or DataArray*) – First quartile of daily total precipitation computed by month. .. warning:: Before computing the percentiles, all the precipitation below 1mm must be filtered out ! Otherwise, the percentiles will include non-wet days. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

cold_and_dry_days (*DataArray*) – Cold and dry days [days] cell_methods: time: sum over days description: {freq} number of days where tas < {tas_per_thresh}th percentile and pr < {pr_per_thresh}th percentile

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [cold_dry_days].

References

`xclim.indicators.atmos._precip.cold_and_wet_days(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', tas_per: Union[DataArray, str] = 'tas_per', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', ds: Dataset = None, **indexer*`) → DataArray`

cold and wet days (realm: atmos)

Returns the total number of days where “cold” and “wet” conditions coincide.

This indicator will check for missing values according to the method “from_context”. Based on indice `cold_and_wet_days()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature values Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **tas_per** (*str or DataArray*) – First quartile of daily mean temperature computed by month. Default : *ds.tas_per*. [Required units : [temperature]]
- **pr_per** (*str or DataArray*) – Third quartile of daily total precipitation computed by month. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

cold_and_wet_days (*DataArray*) – cold and wet days [days] cell_methods: time: sum over days description: {freq} number of days where tas < {tas_per_thresh}th percentile and pr > {pr_per_thresh}th percentile

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [cold_wet_days].

References

`xclim.indicators.atmos._precip.daily_pr_intensity(`*pr: Union[DataArray, str] = 'pr'*, ***, *thresh: str = '1 mm/day'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*`)` → DataArray

Average daily precipitation intensity. (realm: atmos)

Return the average precipitation over wet days.

This indicator will check for missing values according to the method "from_context". Based on indice `daily_pr_intensity()`.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

sdii (*DataArray*) – Average precipitation during wet days (SDII) (lwe_thickness_of_precipitation_amount) [mm/day] description: {freq} Simple Daily Intensity Index (SDII) : {freq} average precipitation for days with daily precipitation over {thresh}. This indicator is also known as the 'Simple Daily Intensity Index' (SDII).

Notes

Let $\mathbf{p} = p_0, p_1, \ldots, p_n$ be the daily precipitation and $thresh$ be the precipitation threshold defining wet days. Then the daily precipitation intensity is defined as

$$\frac{\sum_{i=0}^{n} p_i [p_i \leq thresh]}{\sum_{i=0}^{n} [p_i \leq thresh]}$$

where $[P]$ is 1 if P is true, and 0 if false.

`xclim.indicators.atmos._precip.days_over_precip_doy_thresh(`*pr: Union[DataArray, str] = 'pr', pr_per: Union[DataArray, str] = 'pr_per', *, thresh: str = '1 mm/day', freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of wet days with daily precipitation over a given percentile. (realm: atmos)

Number of days over period where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *days_over_precip_thresh()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – Percentile of wet day precipitation flux. Either computed daily (one value per day of year) or computed over a period (one value per spatial point). Default : *ds.pr_per*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

days_over_precip_doy_thresh (*DataArray*) – Count of days with daily precipitation above the given percentile [days]. (number_of_days_with_lwe_thickness_of_precipitation_amount_above_daily_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with precipitation above the {pr_per_thresh}th daily percentile. Only days with at least {thresh} are counted. A {pr_per_window} day(s) window, centred on each calendar day in the {pr_per_period} period, is used to compute the {pr_per_thresh}th percentile(s).

`xclim.indicators.atmos._precip.days_over_precip_thresh`(*pr: Union[DataArray, str] = 'pr'*, *pr_per: Union[DataArray, str] = 'pr_per'*, ***, *thresh: str = '1 mm/day'*, *freq: str = 'YS'*, *bootstrap: bool = False*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of wet days with daily precipitation over a given percentile. (realm: atmos)

Number of days over period where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *days_over_precip_thresh()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – Percentile of wet day precipitation flux. Either computed daily (one value per day of year) or computed over a period (one value per spatial point). Default : *ds.pr_per*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

days_over_precip_thresh (*DataArray*) – Count of days with daily precipitation above the given percentile [days]. (number_of_days_with_lwe_thickness_of_precipitation_amount_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with precipitation above the {pr_per_thresh}th percentile of {pr_per_period} period. Only days with at least {thresh} are counted.

`xclim.indicators.atmos._precip.days_with_snow`(*prsn: Union[DataArray, str] = 'prsn'*, ***, *low: str = '0 kg m-2 s-1'*, *high: str = '1E6 kg m-2 s-1'*, *freq: str = 'AS-JUL'*, *ds: Dataset = None*, ***indexer*) → DataArray

Days with snowfall (realm: atmos)

Return the number of days where snowfall is within low and high thresholds.

This indicator will check for missing values according to the method "from_context". Based on indice *days_with_snow()*.

Parameters

- **prsn** (*str or DataArray*) – Solid precipitation flux. Default : *ds.prsn*. [Required units : [precipitation]]
- **low** (*quantity (string with units)*) – Minimum threshold solid precipitation flux. Default : 0 kg m-2 s-1. [Required units : [precipitation]]
- **high** (*quantity (string with units)*) – Maximum threshold solid precipitation flux. Default : 1E6 kg m-2 s-1. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

days_with_snow (*DataArray*) – Number of days with solid precipitation flux between low and high thresholds. [days] description: {freq} number of days with solid precipitation flux larger than {low} and smaller or equal to {high}.

References

Matthews, L., Andrey, J., & Picketts, I. (2017). Planning for Winter Road Maintenance in the Context of Climate Change, Weather, Climate, and Society, 9(3), 521-532, https://doi.org/10.1175/WCAS-D-16-0103.1

`xclim.indicators.atmos._precip.drought_code(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', lat: Union[DataArray, str] = 'lat', snd: Optional[Union[DataArray, str]] = None, dc0: Optional[Union[DataArray, str]] = None, season_mask: Optional[Union[DataArray, str]] = None, *, season_method: str | None = None, overwintering: bool = False, dry_start: str | None = None, initial_start_up: bool = True, ds: Dataset = None, **params*`)` → DataArray

Drought code (FWI component). (realm: atmos)

The drought code is part of the Canadian Forest Fire Weather Index System. It is a numeric rating of the average moisture content of organic layers.

This indicator will check for missing values according to the method "skip". Based on indice `drought_code()`.

Parameters

- **tas** (*str or DataArray*) – Noon temperature. Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Rain fall in open over previous 24 hours, at noon. Default : *ds.pr*. [Required units : [precipitation]]
- **lat** (*str or DataArray*) – Latitude coordinate Default : *ds.lat*. [Required units : []]
- **snd** (*str or DataArray, optional*) – Noon snow depth. [Required units : [length]]

- **dc0** (*str or DataArray, optional*) – Initial values of the drought code. [Required units : []]
- **season_mask** (*str or DataArray, optional*) – Boolean mask, True where/when the fire season is active. [Required units : []]
- **season_method** (*{'LA08', None, 'GFWED', 'WF93'}*) – How to compute the start-up and shutdown of the fire season. If "None", no start-ups or shutdowns are computed, similar to the R fwi function. Ignored if *season_mask* is given. Default : None.
- **overwintering** (*boolean*) – Whether to activate DC overwintering or not. If True, either season_method or season_mask must be given. Default : False.
- **dry_start** (*{None, 'GFWED', 'CFS'}*) – Whether to activate the DC and DMC "dry start" mechanism and which method to use. , see `fire_weather_ufunc()`. Default : None.
- **initial_start_up** (*boolean*) – If True (default), grid points where the fire season is active on the first timestep go through a start_up phase for that time step. Otherwise, previous codes must be given as a continuing fire season is assumed for those points. Default : True.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **params** – Any other keyword parameters as defined in *xclim.indices.fwi.fire_weather_ufunc* and in `default_params`. Default : None.

Returns

dc (*DataArray*) – Drought Code (drought_code) description: Numeric rating of the average moisture content of organic layers.

Notes

See https://cwfis.cfs.nrcan.gc.ca/background/dsm/fwi, the module's doc and doc of `fire_weather_ufunc()` for more information.

References

Updated source code for calculating fire danger indexes in the Canadian Forest Fire Weather Index System, Y. Wang, K.R. Anderson, and R.M. Suddaby, INFORMATION REPORT NOR-X-424, 2015.

`xclim.indicators.atmos._precip.dry_days`(*pr: Union[DataArray, str] = 'pr'*, ***, *thresh: str = '0.2 mm/d'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Dry days. (realm: atmos)

The number of days with daily precipitation below threshold.

This indicator will check for missing values according to the method "from_context". Based on indice `dry_days()`.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0.2 mm/d. [Required units : [precipitation]]

- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
dry_days (*DataArray*) – Number of dry days (precip < {thresh}) (number_of_days_with_lwe_thickness_of_precipitation_amount_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with daily precipitation under {thresh}.

Notes

Let PR_{ij} be the daily precipitation at day i of period j. Then counted is the number of days where:

$$\sum PR_{ij} < Threshold[mm/day]$$

`xclim.indicators.atmos._precip.dry_spell_frequency(`*pr: Union[DataArray, str] = 'pr', *, thresh: str = '1.0 mm', window: int = 3, freq: str = 'YS', op: str = 'sum', ds: Dataset = None*`)` → DataArray

Return the number of dry periods of n days and more. (realm: atmos)

Periods during which the accumulated or maximal daily precipitation amount on a window of n days is under threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *dry_spell_frequency()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation amount under which a period is considered dry. The value against which the threshold is compared depends on *op* . Default : 1.0 mm. [Required units : [length]]
- **window** (*number*) – Minimum length of the spells. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **op** (*{'sum', 'max'}*) – Operation to perform on the window. Default is "sum", which checks that the sum of accumulated precipitation over the whole window is less than the threshold. "max" checks that the maximal daily precipitation amount within the window is less than the threshold. This is the same as verifying that each individual day is below the threshold. Default : sum.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
dry_spell_frequency (*DataArray*) – The {freq} number of dry periods of minimum {window} days. description: The {freq} number of dry periods of {window} days and more, during which the {op} precipitation on a window of {window} days is under {thresh}.

`xclim.indicators.atmos._precip.dry_spell_total_length`(*pr: Union[DataArray, str] = 'pr'*, ***, *thresh: str = '1.0 mm'*, *window: int = 3*, *op: str = 'sum'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Total length of dry spells. (realm: atmos)

Total number of days in dry periods of a minimum length, during which the maximum or accumulated precipitation within a window of the same length is under a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *dry_spell_total_length()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Accumulated precipitation value under which a period is considered dry. Default : 1.0 mm. [Required units : [length]]
- **window** (*number*) – Number of days when the maximum or accumulated precipitation is under threshold. Default : 3.
- **op** (*{'sum', 'max'}*) – Reduce operation. Default : sum.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Indexing is done after finding the dry days, but before finding the spells. Default : None.

Returns

dry_spell_total_length (*DataArray*) – The {freq} total number of days in dry periods of minimum {window} days. [days] description: The {freq} number of days in dry periods of {window} days and more, during which the {op}precipitation within windows of {window} days is under {thresh}.

Notes

The algorithm assumes days before and after the timeseries are "wet", meaning that the condition for being considered part of a dry spell is stricter on the edges. For example, with *window=3* and *op='sum'*, the first day of the series is considered part of a dry spell only if the accumulated precipitation within the first 3 days is under the threshold. In comparison, a day in the middle of the series is considered part of a dry spell if any of the three 3-day periods of which it is part are considered dry (so a total of five days are included in the computation, compared to only 3.)

`xclim.indicators.atmos._precip.fire_weather_indexes(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', sfcWind: Union[DataArray, str] = 'sfcWind', hurs: Union[DataArray, str] = 'hurs', lat: Union[DataArray, str] = 'lat', snd: Optional[Union[DataArray, str]] = None, ffmc0: Optional[Union[DataArray, str]] = None, dmc0: Optional[Union[DataArray, str]] = None, dc0: Optional[Union[DataArray, str]] = None, season_mask: Optional[Union[DataArray, str]] = None, *, season_method: str | None = None, overwintering: bool = False, dry_start: str | None = None, initial_start_up: bool = True, ds: Dataset = None, **params*`)` → Tuple[DataArray, DataArray, DataArray, DataArray, DataArray, DataArray]

Fire weather indexes. (realm: atmos)

Computes the 6 fire weather indexes as defined by the Canadian Forest Service: the Drought Code, the Duff-Moisture Code, the Fine Fuel Moisture Code, the Initial Spread Index, the Build Up Index and the Fire Weather Index.

This indicator will check for missing values according to the method "skip". Based on indice *fire_weather_indexes()*.

Parameters

- **tas** (*str or DataArray*) – Noon temperature. Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Rain fall in open over previous 24 hours, at noon. Default : *ds.pr*. [Required units : [precipitation]]
- **sfcWind** (*str or DataArray*) – Noon wind speed. Default : *ds.sfcWind*. [Required units : [speed]]
- **hurs** (*str or DataArray*) – Noon relative humidity. Default : *ds.hurs*. [Required units : []]
- **lat** (*str or DataArray*) – Latitude coordinate Default : *ds.lat*. [Required units : []]
- **snd** (*str or DataArray, optional*) – Noon snow depth, only used if *season_method='LA08'* is passed. [Required units : [length]]
- **ffmc0** (*str or DataArray, optional*) – Initial values of the fine fuel moisture code. [Required units : []]
- **dmc0** (*str or DataArray, optional*) – Initial values of the Duff moisture code. [Required units : []]
- **dc0** (*str or DataArray, optional*) – Initial values of the drought code. [Required units : []]
- **season_mask** (*str or DataArray, optional*) – Boolean mask, True where/when the fire season is active. [Required units : []]
- **season_method** (*{'LA08', None, 'GFWED', 'WF93'}*) – How to compute the start-up and shutdown of the fire season. If "None", no start-ups or shutdowns are

computed, similar to the R fwi function. Ignored if *season_mask* is given. Default : None.

- **overwintering** (*boolean*) – Whether to activate DC overwintering or not. If True, either season_method or season_mask must be given. Default : False.
- **dry_start** (*{None, 'GFWED', 'CFS'}*) – Whether to activate the DC and DMC "dry start" mechanism or not, see `fire_weather_ufunc()`. Default : None.
- **initial_start_up** (*boolean*) – If True (default), gridpoints where the fire season is active on the first timestep go through a start_up phase for that time step. Otherwise, previous codes must be given as a continuing fire season is assumed for those points. Default : True.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **params** – Any other keyword parameters as defined in `fire_weather_ufunc()` and in `default_params`. Default : None.

Returns

- **dc** (*DataArray*) – Drought Code (drought_code) description: Numeric rating of the average moisture content of deep, compact organic layers.
- **dmc** (*DataArray*) – Duff Moisture Code (duff_moisture_code) description: Numeric rating of the average moisture content of loosely compacted organic layers of moderate depth.
- **ffmc** (*DataArray*) – Fine Fuel Moisture Code (fine_fuel_moisture_code) description: Numeric rating of the average moisture content of litter and other cured fine fuels.
- **isi** (*DataArray*) – Initial Spread Index (initial_spread_index) description: Numeric rating of the expected rate of fire spread.
- **bui** (*DataArray*) – Buildup Index (buildup_index) description: Numeric rating of the total amount of fuel available for combustion.
- **fwi** (*DataArray*) – Fire Weather Index (fire_weather_index) description: Numeric rating of fire intensity.

Notes

See https://cwfis.cfs.nrcan.gc.ca/background/dsm/fwi, the module's doc and doc of `fire_weather_ufunc()` for more information.

References

Updated source code for calculating fire danger indexes in the Canadian Forest Fire Weather Index System, Y. Wang, K.R. Anderson, and R.M. Suddaby, INFORMATION REPORT NOR-X-424, 2015.

`xclim.indicators.atmos._precip.first_snowfall`(*prsn: Union[DataArray, str] = 'prsn'*, ***, *thresh: str = '0.5 mm/day'*, *freq: str = 'AS-JUL'*, *ds: Dataset = None*, ***indexer*) → DataArray

First day with solid precipitation above a threshold. (realm: atmos)

Returns the first day of a period where the solid precipitation exceeds a threshold. WARNING: The default *freq* is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *first_snowfall()*.

Parameters

- **prsn** (*str or DataArray*) – Solid precipitation flux. Default : *ds.prsn*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold precipitation flux on which to base evaluation. Default : 0.5 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

first_snowfall (*DataArray*) – Date of first snowfall (day_of_year) description: {freq} first day where the solid precipitation flux exceeded {thresh}

References

Climate Projections for the National Capital Region (2020), Volume 1: Results and Interpretation for Key Climate Indices, Report 193600.00, Prepared for Ottawa by CBCL.

`xclim.indicators.atmos._precip.fraction_over_precip_doy_thresh(`*pr: Union[DataArray, str] = 'pr', pr_per: Union[DataArray, str] = 'pr_per', *, thresh: str = '1 mm/day', freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Fraction of precipitation due to wet days with daily precipitation over a given percentile. (realm: atmos)

Percentage of the total precipitation over period occurring in days where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *fraction_over_precip_thresh()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – Percentile of wet day precipitation flux. Either computed daily (one value per day of year) or computed over a period (one value per spatial point). Default : *ds.pr_per*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and

the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.

- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

fraction_over_precip_doy_thresh (*DataArray*) – Fraction of precipitation over threshold during wet days. description: {freq} fraction of total precipitation due to days with precipitation above {pr_per_thresh}th daily percentile. Only days with at least {thresh} are included in the total. A {pr_per_window} day(s) window, centred on each calendar day in the {pr_per_period} period, is used to compute the {pr_per_thresh}th percentile(s).

`xclim.indicators.atmos._precip.fraction_over_precip_thresh`(*pr: Union[DataArray, str] = 'pr', pr_per: Union[DataArray, str] = 'pr_per', *, thresh: str = '1 mm/day', freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*) → DataArray

Fraction of precipitation due to wet days with daily precipitation over a given percentile. (realm: atmos)

Percentage of the total precipitation over period occurring in days where the precipitation is above a threshold defining wet days and above a given percentile for that day.

This indicator will check for missing values according to the method "from_context". Based on indice *`fraction_over_precip_thresh()`*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **pr_per** (*str or DataArray*) – Percentile of wet day precipitation flux. Either computed daily (one value per day of year) or computed over a period (one value per spatial point). Default : *ds.pr_per*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

fraction_over_precip_thresh (*DataArray*) – Fraction of precipitation over threshold during wet days. description: {freq} fraction of total precipitation due to days with precipitation above {pr_per_thresh}th percentile of {pr_per_period} period. Only days with at least {thresh} are included in the total.

`xclim.indicators.atmos._precip.high_precip_low_temp(`*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, pr_thresh: str = '0.4 mm/d', tas_thresh: str = '-0.2 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with precipitation above threshold and temperature below threshold. (realm: atmos)

Number of days where precipitation is greater or equal to some threshold, and temperatures are colder than some threshold. This can be used for example to identify days with the potential for freezing rain or icing conditions.

This indicator will check for missing values according to the method "from_context". Based on indice *high_precip_low_temp()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Daily mean, minimum or maximum temperature. Default : *ds.tas*. [Required units : [temperature]]
- **pr_thresh** (*quantity (string with units)*) – Precipitation threshold to exceed. Default : 0.4 mm/d. [Required units : [precipitation]]
- **tas_thresh** (*quantity (string with units)*) – Temperature threshold not to exceed. Default : -0.2 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

high_precip_low_temp (*DataArray*) – Count of days with high precipitation and low temperatures. [days] cell_methods: time: sum over days description: {freq} number of days with precipitation above {pr_thresh} and temperature below {tas_thresh}.

`xclim.indicators.atmos._precip.last_snowfall(`*prsn: Union[DataArray, str] = 'prsn', *, thresh: str = '0.5 mm/day', freq: str = 'AS-JUL', ds: Dataset = None, **indexer*`)` → DataArray

Last day with solid precipitation above a threshold. (realm: atmos)

Returns the last day of a period where the solid precipitation exceeds a threshold. WARNING: The default freq is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *last_snowfall()*.

Parameters

- **prsn** (*str or DataArray*) – Solid precipitation flux. Default : *ds.prsn*. [Required units : [precipitation]]

- **thresh** (*quantity (string with units)*) – Threshold precipitation flux on which to base evaluation. Default : 0.5 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

last_snowfall (*DataArray*) – Date of last snowfall (day_of_year) description: {freq} last day where the solid precipitation flux exceeded {thresh}

References

Climate Projections for the National Capital Region (2020), Volume 1: Results and Interpretation for Key Climate Indices, Report 193600.00, Prepared for Ottawa by CBCL.

`xclim.indicators.atmos._precip.liquid_precip_accumulation(`*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Accumulated liquid precipitation. (realm: atmos)

Resample the original daily mean precipitation flux and accumulate over each period. If a daily temperature is provided, the *phase* keyword can be used to sum precipitation of a given phase only. When the temperature is under the provided threshold, precipitation is assumed to be snow, and liquid rain otherwise. This indice is agnostic to the type of daily temperature (tas, tasmax or tasmin) given.

This indicator will check for missing values according to the method "from_context". Based on indice *precip_accumulation()*. With injected parameters: phase=liquid.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean, maximum or minimum daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold of *tas* over which the precipication is assumed to be liquid rain. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

liquidprcptot (*DataArray*) – Total liquid precipitation (lwe_thickness_of_liquid_precipitation_amount) [mm] cell_methods: time: sum over days description: {freq} total {phase} precipitation, estimated as precipitation when temperature >= {thresh}

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$

If tas and phase are given, the corresponding phase precipitation is estimated before computing the accumulation, using one of *snowfall_approximation* or *rain_approximation* with the *binary* method.

`xclim.indicators.atmos._precip.liquid_precip_ratio(`*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', freq: str = 'QS-DEC', ds: Dataset = None, **indexer*`)` → DataArray

Ratio of rainfall to total precipitation. (realm: atmos)

The ratio of total liquid precipitation over the total precipitation. Liquid precipitation is approximated from total precipitation on days where temperature is above a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *liquid_precip_ratio()*. With injected parameters: prsn=None.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature under which precipitation is assumed to be solid. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : QS-DEC.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

liquid_precip_ratio (*DataArray*) – Ratio of rainfall to total precipitation. description: {freq} ratio of rainfall to total precipitation. Rainfall is estimated as precipitation on days where temperature is above {thresh}.

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$

$$PRwet_{ij}$$

`xclim.indicators.atmos._precip.max_1day_precipitation_amount`(*pr: Union[DataArray, str] = 'pr'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Highest 1-day precipitation amount for a period (frequency). (realm: atmos)

Resample the original daily total precipitation temperature series by taking the max over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *max_1day_precipitation_amount()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation values. Default : *ds.pr*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

rx1day (*DataArray*) – maximum 1-day total precipitation (lwe_thickness_of_precipitation_amount) [mm/day] cell_methods: time: maximum over days description: {freq} maximum 1-day total precipitation

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j:

$$PRx_{ij} = max(PR_{ij})$$

`xclim.indicators.atmos._precip.max_n_day_precipitation_amount`(*pr: Union[DataArray, str] = 'pr'*, ***, *window: int = 1*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Highest precipitation amount cumulated over a n-day moving window. (realm: atmos)

Calculate the n-day rolling sum of the original daily total precipitation series and determine the maximum value over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *max_n_day_precipitation_amount()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation values. Default : *ds.pr*. [Required units : [precipitation]]
- **window** (*number*) – Window size in days. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

rx{window}day (*DataArray*) – maximum {window}-day total precipitation (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: maximum over days description: {freq} maximum {window}-day total precipitation.

`xclim.indicators.atmos._precip.max_pr_intensity`(*pr: Union[DataArray, str] = 'pr'*, ***, *window: int = 1*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Highest precipitation intensity over a n-hour moving window. (realm: atmos)

Calculate the n-hour rolling average of the original hourly total precipitation series and determine the maximum value over each period.

This indicator will check for missing values according to the method "from_context". Based on indice *max_pr_intensity()* . Keywords : IDF curves.

Parameters

- **pr** (*str or DataArray*) – Hourly precipitation values. Default : *ds.pr*. [Required units : [precipitation]]
- **window** (*number*) – Window size in hours. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

max_pr_intensity (*DataArray*) – Maximum precipitation intensity over {window}h duration (precipitation) [mm/h] cell_methods: time: max description: {freq} maximum precipitation intensity over rolling {window}h window.

`xclim.indicators.atmos._precip.maximum_consecutive_dry_days`(*pr: Union[DataArray, str] = 'pr'*, ***, *thresh: str = '1 mm/day'*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Maximum number of consecutive dry days. (realm: atmos)

Return the maximum number of consecutive days within the period where precipitation is below a certain threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *maximum_consecutive_dry_days()* .

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold precipitation on which to base evaluation. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

cdd (*DataArray*) – Maximum consecutive dry days (Precip < {thresh}) (number_of_days_with_lwe_thickness_of_precipitation_amount_below_threshold) [days] cell_methods: time: sum over days description: {freq} maximum number of consecutive days with daily precipitation below {thresh}.

Notes

Let $\mathbf{p} = p_0, p_1, \ldots, p_n$ be a daily precipitation series and *thresh* the threshold under which a day is considered dry. Then let $\mathbf{s}$ be the sorted vector of indices i where $[p_i < thresh] \neq [p_{i+1} < thresh]$, that is, the days when the precipitation crosses the threshold. Then the maximum number of consecutive dry days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[p_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indicators.atmos._precip.maximum_consecutive_wet_days`(*pr: Union[DataArray, str] = 'pr', *, thresh: str = '1 mm/day', freq: str = 'YS', ds: Dataset = None*) → DataArray

Consecutive wet days. (realm: atmos)

Returns the maximum number of consecutive wet days.

This indicator will check for missing values according to the method "from_context". Based on indice *maximum_consecutive_wet_days()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold precipitation on which to base evaluation. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

cwd (*DataArray*) – Maximum consecutive wet days (Precip >= {thresh}) (number_of_days_with_lwe_thickness_of_precipitation_amount_at_or_above_threshold) [days] cell_methods: time: sum over days description: {freq} maximum number of consecutive days with daily precipitation over {thresh}.

Notes

Let $\mathbf{x} = x_0, x_1, \ldots, x_n$ be a daily precipitation series and $\mathbf{s}$ be the sorted vector of indices i where $[p_i > thresh] \neq [p_{i+1} > thresh]$, that is, the days when the precipitation crosses the *wet day* threshold. Then the maximum number of consecutive wet days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[x_{s_j} > 0^\circ C]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indicators.atmos._precip.precip_accumulation`(*pr: Union[DataArray, str] = 'pr', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Accumulated total precipitation (solid and liquid) (realm: atmos)

Resample the original daily mean precipitation flux and accumulate over each period. If a daily temperature is provided, the *phase* keyword can be used to sum precipitation of a given phase only.

When the temperature is under the provided threshold, precipitation is assumed to be snow, and liquid rain otherwise. This indice is agnostic to the type of daily temperature (tas, tasmax or tasmin) given.

This indicator will check for missing values according to the method "from_context". Based on indice *precip_accumulation()*. With injected parameters: tas=None, phase=None.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold of *tas* over which the precipitation is assumed to be liquid rain. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

prcptot (*DataArray*) – Total precipitation (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum over days description: {freq} total precipitation

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$

If tas and phase are given, the corresponding phase precipitation is estimated before computing the accumulation, using one of *snowfall_ approximation* or *rain_ approximation* with the *binary* method.

`xclim.indicators.atmos._precip.rain_on_frozen_ground_days(`*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, thresh: str = '1 mm/d', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of rain on frozen ground events. (realm: atmos)

Number of days with rain above a threshold after a series of seven days below freezing temperature. Precipitation is assumed to be rain when the temperature is above 0℃.

This indicator will check for missing values according to the method "from_context". Based on indice *rain_on_frozen_ground_days()*.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Precipitation threshold to consider a day as a rain event. Default : 1 mm/d. [Required units : [precipitation]]

- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

rain_frzgr (*DataArray*) – Number of rain on frozen ground days (number_of_days_with_lwe_thickness_of_precipitation_amount_above_threshold) [days] description: {freq} number of days with rain above {thresh} after a series of seven days with average daily temperature below 0℃. Precipitation is assumed to be rain when thedaily average temperature is above 0℃.

Notes

Let PR_i be the mean daily precipitation and TG_i be the mean daily temperature of day i. Then for a period j, rain on frozen grounds days are counted where:

$$PR_i > Threshold[mm]$$

and where

$$TG_i 0$$

is true for continuous periods where $i 7$

`xclim.indicators.atmos._precip.rprctot(`*pr: Union[DataArray, str] = 'pr', prc: Union[DataArray, str] = 'prc', *, thresh: str = '1.0 mm/day', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Proportion of accumulated precipitation arising from convective processes. (realm: atmos)

Return the proportion of total accumulated precipitation due to convection on days with total precipitation exceeding a specified threshold during the given period.

This indicator will check for missing values according to the method "from_context". Based on indice *rprctot()*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **prc** (*str or DataArray*) – Daily convective precipitation. Default : *ds.prc*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1.0 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

rprctot (*DataArray*) – The proportion of the total precipitation accounted for by convective precipitation for each period. cell_methods: time: sum description: Proportion of accumulated precipitation arising from convective processes.

`xclim.indicators.atmos._precip.solid_precip_accumulation`(*pr: Union[DataArray, str] = 'pr', tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Accumulated solid precipitation. (realm: atmos)

Resample the original daily mean precipitation flux and accumulate over each period. If a daily temperature is provided, the *phase* keyword can be used to sum precipitation of a given phase only. When the temperature is under the provided threshold, precipitation is assumed to be snow, and liquid rain otherwise. This indice is agnostic to the type of daily temperature (tas, tasmax or tasmin) given.

This indicator will check for missing values according to the method "from_context". Based on indice *precip_accumulation()*. With injected parameters: phase=solid.

Parameters

- **pr** (*str or DataArray*) – Mean daily precipitation flux. Default : *ds.pr*. [Required units : [precipitation]]
- **tas** (*str or DataArray*) – Mean, maximum or minimum daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold of *tas* over which the precipication is assumed to be liquid rain. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

solidprcptot (*DataArray*) – Total solid precipitation (lwe_thickness_of_snowfall_amount) [mm] cell_methods: time: sum over days description: {freq} total solid precipitation, estimated as precipitation when temperature < {thresh}

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$

If tas and phase are given, the corresponding phase precipitation is estimated before computing the accumulation, using one of *snowfall_approximation* or *rain_approximation* with the *binary* method.

`xclim.indicators.atmos._precip.warm_and_dry_days(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', tas_per: Union[DataArray, str] = 'tas_per', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

warm and dry days (realm: atmos)

Returns the total number of days where "warm" and "Dry" conditions coincide.

This indicator will check for missing values according to the method "from_context". Based on indice *warm_and_dry_days()* .

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature values Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **tas_per** (*str or DataArray*) – Third quartile of daily mean temperature computed by month. Default : *ds.tas_per*. [Required units : [temperature]]
- **pr_per** (*str or DataArray*) – First quartile of daily total precipitation computed by month. .. warning:: Before computing the percentiles, all the precipitation below 1mm must be filtered out ! Otherwise, the percentiles will include non-wet days. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

warm_and_dry_days (*DataArray*) – warm and dry days [days] cell_methods: time: sum over days description: {freq} number of days where tas > {tas_per_thresh}th percentile and pr < {pr_per_thresh}th percentile

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [warm_dry_days].

References

`xclim.indicators.atmos._precip.warm_and_wet_days(`*tas: Union[DataArray, str] = 'tas', pr: Union[DataArray, str] = 'pr', tas_per: Union[DataArray, str] = 'tas_per', pr_per: Union[DataArray, str] = 'pr_per', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

warm and wet days (realm: atmos)

Returns the total number of days where "warm" and "wet" conditions coincide.

This indicator will check for missing values according to the method "from_context". Based on indice *warm_and_wet_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature values Default : *ds.tas*. [Required units : [temperature]]
- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **tas_per** (*str or DataArray*) – Third quartile of daily mean temperature computed by month. Default : *ds.tas_per*. [Required units : [temperature]]
- **pr_per** (*str or DataArray*) – Third quartile of daily total precipitation computed by month. .. warning:: Before computing the percentiles, all the precipitation below 1mm must be filtered out ! Otherwise, the percentiles will include non-wet days. Default : *ds.pr_per*. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

warm_and_wet_days (*DataArray*) – warm and wet days [days] cell_methods: time: sum over days description: {freq} number of days where tas > {tas_per_thresh}th percentile and pr > {pr_per_thresh}th percentile

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [warm_wet_days].

References

`xclim.indicators.atmos._precip.wet_precip_accumulation(`*pr: Union[DataArray, str] = 'pr', *, thresh: str = '1 mm/day', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Accumulated total precipitation (solid and liquid) during wet days (realm: atmos)

This indicator will check for missing values according to the method "from_context". Based on indice *`prcptot()`*.

Parameters

- **pr** (*str or DataArray*) – Total precipitation flux [mm d-1], [mm week-1], [mm month-1] or similar. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Threshold over which precipitation starts being cumulated. Default : 1 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

wet_prcptot (*DataArray*) – Total precipitation (lwe_thickness_of_precipitation_amount) [mm] cell_methods: time: sum over days description: {freq} total precipitation over wet days, defined as days where precipitation exceeds {thresh}.

`xclim.indicators.atmos._precip.wetdays(`*pr: Union[DataArray, str] = 'pr', *, thresh: str = '1.0 mm/day', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Wet days. (realm: atmos)

Return the total number of days during period with precipitation over threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *`wetdays()`*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1.0 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

wetdays (*DataArray*) – Number of wet days (precip >= {thresh}) (number_of_days_with_lwe_thickness_of_precipitation_amount_at_or_above_threshold)

[days] cell_methods: time: sum over days description: {freq} number of days with daily precipitation over {thresh}.

`xclim.indicators.atmos._precip.wetdays_prop`(*pr: Union[DataArray, str] = 'pr'*, **, *thresh: str = '1.0 mm/day'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Proportion of wet days. (realm: atmos)

Return the proportion of days during period with precipitation over threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *`wetdays_prop()`*.

Parameters

- **pr** (*str or DataArray*) – Daily precipitation. Default : *ds.pr*. [Required units : [precipitation]]
- **thresh** (*quantity (string with units)*) – Precipitation value over which a day is considered wet. Default : 1.0 mm/day. [Required units : [precipitation]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

wetdays_prop (*DataArray*) – Proportion of wet days (precip >= {thresh}) [1] cell_methods: time: sum over days description: {freq} proportion of days with precipitation over {thresh}.

xclim.indicators.atmos._synoptic module

Synoptic indicator definitions.

`xclim.indicators.atmos._synoptic.jetstream_metric_woollings`(*ua: Union[DataArray, str] = 'ua'*, ***, *ds: Dataset = None*) → Tuple[DataArray, DataArray]

Strength and latitude of jetstream. (realm: atmos)

Identify latitude and strength of maximum smoothed zonal wind speed in the region from 15 to 75°N and -60 to 0°E, using the formula outlined in ([Woollings2010]).

Based on indice *`jetstream_metric_woollings()`*.

Parameters

- **ua** (*str or DataArray*) – Eastward wind component (u) at between 750 and 950 hPa. Default : *ds.ua*. [Required units : [speed]]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

- **jetlat** (*DataArray*) – Latitude of maximum smoothed zonal wind speed [degrees_North] description: Daily latitude of maximum smoothed zonal wind speed
- **jetstr** (*DataArray*) – Maximum strength of smoothed zonal wind speed [m s-1] description: Daily maximum strength of smoothed zonal wind speed

References

xclim.indicators.atmos._temperature module

Temperature indicator definitions.

`xclim.indicators.atmos._temperature.biologically_effective_degree_days(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', lat: Union[DataArray, str] = 'lat', *, thresh_tasmin: str = '10 degC', low_dtr: str = '10 degC', high_dtr: str = '13 degC', max_daily_degree_days: str = '9 degC', start_date: DayOfYearStr = '04-01', end_date: DayOfYearStr = '11-01', freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Biologically effective growing degree days. (realm: atmos)

Growing-degree days with a base of 10°C and an upper limit of 19°C and adjusted for latitudes between 40°N and 50°N for April to October (Northern Hemisphere; October to April in Southern Hemisphere). A temperature range adjustment also promotes small and large swings in daily temperature range. Used as a heat-summation metric in viticulture agroclimatology.

This indicator will check for missing values according to the method "from_context". Based on indice *`biologically_effective_degree_days()`*. With injected parameters: method=gladstones.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **lat** (*str or DataArray*) – Latitude coordinate. Default : *ds.lat*. [Required units : []]
- **thresh_tasmin** (*quantity (string with units)*) – The minimum temperature threshold. Default : 10 degC. [Required units : [temperature]]
- **low_dtr** (*quantity (string with units)*) – The lower bound for daily temperature range adjustment (default: 10°C). Default : 10 degC. [Required units : [temperature]]
- **high_dtr** (*quantity (string with units)*) – The higher bound for daily temperature range adjustment (default: 13°C). Default : 13 degC. [Required units : [temperature]]

- **max_daily_degree_days** (*quantity (string with units)*) – The maximum amount of biologically effective degrees days that can be summed daily. Default : 9 degC. [Required units : [temperature]]
- **start_date** (*date (string, MM-DD)*) – The hemisphere-based start date to consider (north = April, south = October). Default : 04-01.
- **end_date** (*date (string, MM-DD)*) – The hemisphere-based start date to consider (north = October, south = April). This date is non-inclusive. Default : 11-01.
- **freq** (*offset alias (string)*) – Resampling frequency (default: "YS"; For Southern Hemisphere, should be "AS-JUL"). Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

bedd (*DataArray*) – Biologically effective degree days computed with {method} formula (Summation of min((max((Tmin + Tmax)/2 - {thresh_tasmin}, 0) * k) + TR_adg, 9°C), for days between {start_date} and {end_date}). [K days] description: Heat-summation index for agroclimatic suitability estimation, developed specifically for viticulture. Considers daily Tmin and Tmax with a base of {thresh_tasmin} between 1 April and 31 October, with a maximum daily value for degree days (typically 9°C). It also integrates a modification coefficient for latitudes between 40°N and 50°N as well as swings in daily temperature range. comment: Original formula published in Gladstones, 1992.

Notes

The tasmax ceiling of 19°C is assumed to be the max temperature beyond which no further gains from daily temperature occur. Indice originally published in [Gladstones1992].

Let TX_i and TN_i be the daily maximum and minimum temperature at day i, lat the latitude of the point of interest, $degdays_{max}$ the maximum amount of degrees that can be summed per day (typically, 9). Then the sum of daily biologically effective growing degree day (BEDD) units between 1 April and 31 October is:

$$BEDD_i = \sum_{i=\text{April 1}}^{\text{October 31}} min\left(\left(max\left(\frac{TX_i + TN_i)}{2} - 10, 0\right) * k\right) + TR_{adj}, degdays_{max}\right)$$

$$TR_{adj} = f(TX_i, TN_i) = \begin{cases} 0.25(TX_i - TN_i - 13), & \text{if } (TX_i - TN_i) > 13 \\ 0, & \text{if } 10 < (TX_i - TN_i) < 13 \\ 0.25(TX_i - TN_i - 10), & \text{if } (TX_i - TN_i) < 10 \end{cases}$$

$$k = f(lat) = 1 + \left(\frac{|lat|}{50} * 0.06, \text{if } 40 < |lat| < 50, \text{else } 0\right)$$

A second version of the BEDD (*method="icclim"*) does not consider TR_{adj} and k and employs a different end date (30 September) ([ECAD]). The simplified formula is as follows:

$$BEDD_i = \sum_{i=\text{April 1}}^{\text{September 30}} min\left(max\left(\frac{TX_i + TN_i)}{2} - 10, 0\right), degdays_{max}\right)$$

References

`xclim.indicators.atmos._temperature.cold_spell_days(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '-10 degC', window: int = 5, freq: str = 'AS-JUL', ds: Dataset = None*`)` → DataArray

Cold spell days. (realm: atmos)

The number of days that are part of cold spell events, defined as a sequence of consecutive days with mean daily temperature below a threshold in °C.

This indicator will check for missing values according to the method "from_context". Based on indice `cold_spell_days()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature below which a cold spell begins. Default : -10 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature below threshold to qualify as a cold spell. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

cold_spell_days (*DataArray*) – Number of days part of a cold spell (cold_spell_days) [days] description: {freq} number of days that are part of a cold spell, defined as {window} or more consecutive days with mean daily temperature below {thresh}.

Notes

Let T_i be the mean daily temperature on day i, the number of cold spell days during period ϕ is given by

$$\sum_{i\in\phi}\prod_{j=i}^{i+5}[T_j < thresh]$$

where $[P]$ is 1 if P is true, and 0 if false.

`xclim.indicators.atmos._temperature.cold_spell_duration_index(`*tasmin: Union[DataArray, str] = 'tasmin', tasmin_per: Union[DataArray, str] = 'tasmin_per', *, window: int = 6, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None*`)` → DataArray

Cold spell duration index. (realm: atmos)

Number of days with at least *window* consecutive days where the daily minimum temperature is below the *tasmin_per* percentiles.

This indicator will check for missing values according to the method "from_context". Based on indice `cold_spell_duration_index()`.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmin_per** (*str or DataArray*) – nth percentile of daily minimum temperature with *dayofyear* coordinate. Default : *ds.tasmin_per*. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature below threshold to qualify as a cold spell. Default : 6.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

csdi_{window} (*DataArray*) – Number of days part of a percentile-defined cold spell (cold_spell_duration_index) [days] description: {freq} number of days with at least {window} consecutive days where the daily minimum temperature is below the {tasmin_per_thresh}th percentile(s). A {tasmin_per_window} day(s) window, centred on each calendar day in the {tasmin_per_period} period, is used to compute the {tasmin_per_thresh}th percentile(s).

Notes

Let TN_i be the minimum daily temperature for the day of the year i and $TN10_i$ the 10th percentile of the minimum daily temperature over the 1961-1990 period for day of the year i, the cold spell duration index over period ϕ is defined as:

$$\sum_{i\in\phi} \prod_{j=i}^{i+6} [TN_j < TN10_j]$$

where $[P]$ is 1 if P is true, and 0 if false.

References

From the Expert Team on Climate Change Detection, Monitoring and Indices (ETCCDMI).

`xclim.indicators.atmos._temperature.cold_spell_frequency(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '-10 degC', window: int = 5, freq: str = 'AS-JUL', ds: Dataset = None*) → DataArray

Cold spell frequency. (realm: atmos)

The number of cold spell events, defined as a sequence of consecutive days with mean daily temperature below a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice `cold_spell_frequency()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature below which a cold spell begins. Default : -10 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature below threshold to qualify as a cold spell. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

cold_spell_frequency (*DataArray*) – Number of cold spell events (cold_spell_frequency) description: {freq} number cold spell events, defined as {window} or more consecutive days with mean daily temperature below {thresh}.

`xclim.indicators.atmos._temperature.consecutive_frost_days(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '0.0 degC', freq: str = 'AS-JUL', ds: Dataset = None*) → DataArray

Maximum number of consecutive frost days (Tn < 0℃). (realm: atmos)

The maximum number of consecutive days within the period where the temperature is under a certain threshold (default: 0°C). WARNING: The default freq value is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *maximum_consecutive_frost_days()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature. Default : 0.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

consecutive_frost_days (*DataArray*) – Maximum number of consecutive days with Tmin < {thresh} (spell_length_of_days_with_air_temperature_below_threshold) [days] cell_methods: time: maximum over days description: {freq} maximum number of consecutive days with minimum daily temperature below {thresh}.

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily minimum temperature series and *thresh* the threshold below which a day is considered a frost day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i < thresh] \neq [t_{i+1} < thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive frost free days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indicators.atmos._temperature.cool_night_index(`*tasmin: Union[DataArray, str] = 'tasmin', lat: Union[DataArray, str] = 'lat', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Cool Night Index. (realm: atmos)

A night coolness variable which takes into account the mean minimum night temperatures during the month when ripening usually occurs beyond the ripening period.

This indicator will check for missing values according to the method "from_context". Based on indice *cool_night_index()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **lat** (*str or DataArray*) – Latitude coordinate. Default : *ds.lat*. [Required units : []]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

cool_night_index (*DataArray*) – cool night index [degC] cell_methods: time: mean over days description: Mean minimum temperature for September (northern hemisphere) or March (southern hemisphere).

Notes

Given that this indice only examines September and March months, it is possible to send in DataArrays containing only these timesteps. Users should be aware that due to the missing values checks in wrapped Indicators, datasets that are missing several months will be flagged as invalid. This check can be ignored by setting the following context:

References

`xclim.indicators.atmos._temperature.cooling_degree_days(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '18.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Cooling degree days. (realm: atmos)

Sum of degree days above the temperature threshold at which spaces are cooled.

This indicator will check for missing values according to the method "from_context". Based on indice *cooling_degree_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Temperature threshold above which air is cooled. Default : 18.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.

- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
cooling_degree_days (*DataArray*) – Cooling degree days (Tmean > {thresh}) (integral_of_air_temperature_excess_wrt_time) [K days] cell_methods: time: sum over days description: {freq} cooling degree days above {thresh}.

Notes

Let x_i be the daily mean temperature at day i. Then the cooling degree days above temperature threshold *thresh* over period ϕ is given by:

$$\sum_{i\in\phi}(x_i - thresh[x_i > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false.

`xclim.indicators.atmos._temperature.daily_freezethaw_cycles(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '0 degC', thresh_tasmax: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Statistics of consecutive diurnal temperature swing events. (realm: atmos)

A diurnal swing of max and min temperature event is when Tmax > thresh_tasmax and Tmin <= thresh_tasmin. This indice finds all days that constitute these events and computes statistics over the length and frequency of these events.

This indicator will check for missing values according to the method "from_context". Based on indice *multiday_temperature_swing()*. With injected parameters: window=1, op=sum.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The temperature threshold needed to trigger a freeze event. Default : 0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The temperature threshold needed to trigger a thaw event. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

dlyfrzthw (*DataArray*) – daily freezethaw cycles [days] description: {freq} number of days with a diurnal freeze-thaw cycle : Tmax > {thresh_tasmax} and Tmin <= {thresh_tasmin}.

Notes

Let TX_i be the maximum temperature at day i and TN_i be the daily minimum temperature at day i. Then freeze thaw spells during a given period are consecutive days where:

$$TX_i > 0 \land TN_i < 0$$

This indice returns a given statistic of the found lengths, optionally dropping those shorter than the *window* argument. For example, *window=1* and *op='sum'* returns the same value as *daily_freezethaw_cycles()*.

`xclim.indicators.atmos._temperature.daily_temperature_range(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Mean of daily temperature range. (realm: atmos)

The mean difference between the daily maximum temperature and the daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *daily_temperature_range()*. With injected parameters: op=mean.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

dtr (*DataArray*) – Mean Diurnal Temperature Range (air_temperature) [K] cell_methods: time range within days time: mean over days description: {freq} mean diurnal temperature range.

Notes

For a default calculation using *op='mean'* :

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the mean diurnal temperature range in period j is:

$$DTR_j = \frac{\sum_{i=1}^{I}(TX_{ij} - TN_{ij})}{I}$$

`xclim.indicators.atmos._temperature.daily_temperature_range_variability(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Mean absolute day-to-day variation in daily temperature range. (realm: atmos)

Mean absolute day-to-day variation in daily temperature range.

This indicator will check for missing values according to the method "from_context". Based on indice `daily_temperature_range_variability()`.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

dtrvar (*DataArray*) – Mean Diurnal Temperature Range Variability (air_temperature) [K] cell_methods: time range within days time: difference over days time: mean over days description: {freq} mean diurnal temparature range variability (defined as the average day-to-day variation in daily temperature range for the given time period)

Notes

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then calculated is the absolute day-to-day differences in period j is:

$$vDTR_j = \frac{\sum_{i=2}^{I} |(TX_{ij} - TN_{ij}) - (TX_{i-1,j} - TN_{i-1,j})|}{I}$$

`xclim.indicators.atmos._temperature.degree_days_exceedance_date(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', sum_thresh: str = '25 K days', op: str = '>', after_date: DayOfYearStr = None, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Degree days exceedance date. (realm: atmos)

Day of year when the sum of degree days exceeds a threshold. Degree days are computed above or below a given temperature threshold.

This indicator will check for missing values according to the method "from_context". Based on indice `degree_days_exceedance_date()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base degree days evaluation. Default : 0 degC. [Required units : [temperature]]
- **sum_thresh** (*quantity (string with units)*) – Threshold of the degree days sum. Default : 25 K days. [Required units : K days]
- **op** (*{'<=', 'lt', '<', 'ge', '>=', 'gt', 'le', '>'}*) – If equivalent to '>', degree days are computed as *tas - thresh* and if equivalent to '<', they are computed as *thresh - tas*. Default : >.
- **after_date** (*date (string, MM-DD)*) – Date at which to start the cumulative sum. In "mm-dd" format, defaults to the start of the sampling period. Default : None.
- **freq** (*offset alias (string)*) – Resampling frequency. If *after_date* is given, *freq* should be annual. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

degree_days_exceedance_date (*DataArray*) – Day of year when cumulative degree days exceed {sum_thresh}. (day_of_year) description: Day of year when the integral of degree days (tmean {op} {thresh}) exceeds {sum_thresh}, the cumulative sum starts on {after_date}.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j, T is the reference threshold and ST is the sum threshold. Then, starting at day :math:i_0:, the degree days exceedance date is the first day k such that

$$\begin{cases}ST < \sum_{i=i_0}^{k} \max(TG_{ij} - T, 0) & \text{if } op \text{ is '>'}\\ ST < \sum_{i=i_0}^{k} \max(T - TG_{ij}, 0) & \text{if } op \text{ is '<'}\end{cases}$$

The resulting k is expressed as a day of year.

Cumulated degree days have numerous applications including plant and insect phenology. See https://en.wikipedia.org/wiki/Growing_degree-day for examples.

`xclim.indicators.atmos._temperature.extreme_temperature_range(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Extreme intra-period temperature range. (realm: atmos)

The maximum of max temperature (TXx) minus the minimum of min temperature (TNn) for the given time period.

This indicator will check for missing values according to the method "from_context". Based on indice *extreme_temperature_range()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

etr (*DataArray*) – Intra-period Extreme Temperature Range (air_temperature) [K] description: {freq} range between the maximum of daily max temperature (tx_max) and the minimum of daily min temperature (tn_min)

Notes

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the extreme temperature range in period j is:

$$ETR_j = max(TX_{ij}) - min(TN_{ij})$$

`xclim.indicators.atmos._temperature.fire_season`(*tas: Union[DataArray, str] = 'tas', snd: Optional[Union[DataArray, str]] = None, *, method: str = 'WF93', freq: str | None = None, temp_start_thresh: str = '12 degC', temp_end_thresh: str = '5 degC', temp_condition_days: int = 3, snow_condition_days: int = 3, snow_thresh: str = '0.01 m', ds: Dataset = None*) → DataArray

Fire season mask. (realm: atmos)

Binary mask of the active fire season, defined by conditions on consecutive daily temperatures and, optionally, snow depths.

Based on indice `fire_season()`.

Parameters

- **tas** (*str or DataArray*) – Daily surface temperature, cffdrs recommends using maximum daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **snd** (*str or DataArray, optional*) – Snow depth, used with method == 'LA08'. [Required units : [length]]
- **method** (*{'LA08', 'GFWED', 'WF93'}*) – Which method to use. "LA08" and "GFWED" need the snow depth. Default : WF93.
- **freq** (*offset alias (string)*) – If given only the longest fire season for each period defined by this frequency, Every "seasons" are returned if None, including the short shoulder seasons. Default : None.
- **temp_start_thresh** (*quantity (string with units)*) – Minimal temperature needed to start the season. Default : 12 degC. [Required units : [temperature]]
- **temp_end_thresh** (*quantity (string with units)*) – Maximal temperature needed to end the season. Default : 5 degC. [Required units : [temperature]]
- **temp_condition_days** (*number*) – Number of days with temperature above or below the thresholds to trigger a start or an end of the fire season. Default : 3.
- **snow_condition_days** (*number*) – Parameters for the fire season determination. See `fire_season()`. Temperature is in degC, snow in m. The *snow_thresh* parameters is also used when *dry_start* is set to "GFWED". Default : 3.
- **snow_thresh** (*quantity (string with units)*) – Minimal snow depth level to end a fire season, only used with method "LA08". Default : 0.01 m. [Required units : [length]]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

fire_season (*DataArray*) – Fire season mask description: Fire season mask, computed with method {method}.

References

[Wotton&Flannigan1993]_

[Lawson&Armitage2008]_

`xclim.indicators.atmos._temperature.first_day_above(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '0 degC', after_date: DayOfYearStr = '01-01', window: int = 1, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

First day of temperatures superior to a threshold temperature. (realm: atmos)

Returns first day of period where a temperature is superior to a threshold over a given number of days, limited to a starting calendar date.

This indicator will check for missing values according to the method "from_context". Based on indice *first_day_above()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **after_date** (*date (string, MM-DD)*) – Date of the year after which to look for the first event. Should have the format '%m-%d'. Default : 01-01.
- **window** (*number*) – Minimum number of days with temperature above threshold needed for evaluation. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

first_day_above (*DataArray*) – First day of year with temperature above {thresh} (day_of_year) description: First day of year with temperature above {thresh} for at least {window} days.

`xclim.indicators.atmos._temperature.first_day_below(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '0 degC', after_date: DayOfYearStr = '07-01', window: int = 1, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

First day of temperatures inferior to a threshold temperature. (realm: atmos)

Returns first day of period where a temperature is inferior to a threshold over a given number of days, limited to a starting calendar date.

This indicator will check for missing values according to the method "from_context". Based on indice *first_day_below()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]

- **after_date** (*date (string, MM-DD)*) – Date of the year after which to look for the first frost event. Should have the format '%m-%d'. Default : 07-01.
- **window** (*number*) – Minimum number of days with temperature below threshold needed for evaluation. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

first_day_below (*DataArray*) – First day of year with temperature below {thresh} (day_of_year) description: First day of year with temperature below {thresh} for at least {window} days.

`xclim.indicators.atmos._temperature.freezethaw_spell_frequency(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '0 degC', thresh_tasmax: str = '0 degC', window: int = 1, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Frequency of freeze-thaw spells (realm: atmos)

A diurnal swing of max and min temperature event is when Tmax > thresh_tasmax and Tmin <= thresh_tasmin. This indice finds all days that constitute these events and computes statistics over the length and frequency of these events.

This indicator will check for missing values according to the method "from_context". Based on indice *multiday_temperature_swing()*. With injected parameters: op=count.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The temperature threshold needed to trigger a freeze event. Default : 0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The temperature threshold needed to trigger a thaw event. Default : 0 degC. [Required units : [temperature]]
- **window** (*number*) – The minimal length of spells to be included in the statistics. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

freezethaw_spell_frequency (*DataArray*) – {freq} number of freeze-thaw spells. [days] description: {freq} number of freeze-thaw spells: Tmax > {thresh_tasmax} and Tmin <= {thresh_tasmin} for at least {window} consecutive day(s).

Notes

Let TX_i be the maximum temperature at day i and TN_i be the daily minimum temperature at day i. Then freeze thaw spells during a given period are consecutive days where:

$$TX_i > 0 \land TN_i < 0$$

This indice returns a given statistic of the found lengths, optionally dropping those shorter than the *window* argument. For example, *window=1* and *op='sum'* returns the same value as `daily_freezethaw_cycles()`.

`xclim.indicators.atmos._temperature.freezethaw_spell_max_length(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '0 degC', thresh_tasmax: str = '0 degC', window: int = 1, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Maximal length of freeze-thaw spells. (realm: atmos)

A diurnal swing of max and min temperature event is when Tmax > thresh_tasmax and Tmin <= thresh_tasmin. This indice finds all days that constitute these events and computes statistics over the length and frequency of these events.

This indicator will check for missing values according to the method "from_context". Based on indice `multiday_temperature_swing()`. With injected parameters: op=max.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The temperature threshold needed to trigger a freeze event. Default : 0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The temperature threshold needed to trigger a thaw event. Default : 0 degC. [Required units : [temperature]]
- **window** (*number*) – The minimal length of spells to be included in the statistics. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

freezethaw_spell_max_length (*DataArray*) – {freq} maximal length of freeze-thaw spells. [days] description: {freq} maximal length of freeze-thaw spells: Tmax > {thresh_tasmax} and Tmin <= {thresh_tasmin} for at least {window} consecutive day(s).

Notes

Let TX_i be the maximum temperature at day i and TN_i be the daily minimum temperature at day i. Then freeze thaw spells during a given period are consecutive days where:

$$TX_i > 0 \wedge TN_i < 0$$

This indice returns a given statistic of the found lengths, optionally dropping those shorter than the *window* argument. For example, *window=1* and *op='sum'* returns the same value as `daily_freezethaw_cycles()`.

`xclim.indicators.atmos._temperature.freezethaw_spell_mean_length(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '0 degC', thresh_tasmax: str = '0 degC', window: int = 1, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Averge length of freeze-thaw spells. (realm: atmos)

A diurnal swing of max and min temperature event is when Tmax > thresh_tasmax and Tmin <= thresh_tasmin. This indice finds all days that constitute these events and computes statistics over the length and frequency of these events.

This indicator will check for missing values according to the method "from_context". Based on indice `multiday_temperature_swing()`. With injected parameters: op=mean.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The temperature threshold needed to trigger a freeze event. Default : 0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The temperature threshold needed to trigger a thaw event. Default : 0 degC. [Required units : [temperature]]
- **window** (*number*) – The minimal length of spells to be included in the statistics. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

freezethaw_spell_mean_length (*DataArray*) – {freq} average length of freeze-thaw spells. [days] description: {freq} average length of freeze-thaw spells: Tmax > {thresh_tasmax} and Tmin <= {thresh_tasmin} for at least {window} consecutive day(s).

Notes

Let TX_i be the maximum temperature at day i and TN_i be the daily minimum temperature at day i. Then freeze thaw spells during a given period are consecutive days where:

$$TX_i > 0 \wedge TN_i < 0$$

This indice returns a given statistic of the found lengths, optionally dropping those shorter than the *window* argument. For example, *window=1* and *op='sum'* returns the same value as `daily_freezethaw_cycles()`.

`xclim.indicators.atmos._temperature.freezing_degree_days(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Heating degree days. (realm: atmos)

Sum of degree days below the temperature threshold at which spaces are heated.

This indicator will check for missing values according to the method "from_context". Based on indice `heating_degree_days()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

freezing_degree_days (*DataArray*) – Freezing degree days (Tmean < {thresh}) (integral_of_air_temperature_deficit_wrt_time) [K days] cell_methods: time: sum over days description: {freq} freezing degree days below {thresh}.

Notes

This index intentionally differs from its ECA&D equivalent: HD17. In HD17, values below zero are not clipped before the sum. The present definition should provide a better representation of the energy demand for heating buildings to the given threshold.

Let TG_{ij} be the daily mean temperature at day i of period j. Then the heating degree days are:

$$HD17_j = \sum_{i=1}^{I} (17 - TG_{ij}) | TG_{ij} < 17)$$

`xclim.indicators.atmos._temperature.freshet_start(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', window: int = 5, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

First day consistently exceeding threshold temperature. (realm: atmos)

Returns first day of period where a temperature threshold is exceeded over a given number of days.

This indicator will check for missing values according to the method "from_context". Based on indice `freshet_start()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold needed for evaluation. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

freshet_start (*DataArray*) – Day of year of spring freshet start (day_of_year) description: Day of year of spring freshet start, defined as the first day a temperature threshold of {thresh} is exceeded for at least {window} days.

Notes

Let x_i be the daily mean temperature at day of the year i for values of i going from 1 to 365 or 366. The start date of the freshet is given by the smallest index i for which

$$\prod_{j=i}^{i+w} [x_j > thresh]$$

is true, where w is the number of days the temperature threshold should be exceeded, and $[P]$ is 1 if P is true, and 0 if false.

`xclim.indicators.atmos._temperature.frost_days(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Frost days index. (realm: atmos)

Number of days where daily minimum temperatures are below a threshold temperature.

This indicator will check for missing values according to the method "from_context". Based on indice `frost_days()`.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Freezing temperature. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
frost_days (*DataArray*) – Number of frost days (Tmin < {thresh}) (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with minimum daily temperature below {thresh}.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j and :math`TT` the threshold. Then counted is the number of days where:

$$TN_{ij} < TT$$

`xclim.indicators.atmos._temperature.frost_free_season_end(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '0 degC', mid_date: DayOfYearStr = '07-01', window: int = 5, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

End of the frost free season. (realm: atmos)

Day of the year of the start of a sequence of days with minimum temperatures consistently below a threshold, after a period with minimum temperatures consistently above the same threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *`frost_free_season_end()`*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **mid_date** (*date (string, MM-DD)*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'. Default : 07-01.
- **window** (*number*) – Minimum number of days with temperature below threshold needed for evaluation. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
frost_free_season_end (*DataArray*) – Day of year of frost free season end (day_of_year) description: Day of year of end of frost free season, defined as the first day minimum temperatures below a threshold of {thresh}, after a run of days above this threshold, for at least {window} days.

`xclim.indicators.atmos._temperature.frost_free_season_length(`*tasmin: Union[DataArray, str] = 'tasmin', *, window: int = 5, mid_date: DayOfYearStr | None = '07-01', thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Frost free season length. (realm: atmos)

The number of days between the first occurrence of at least N (def: 5) consecutive days with minimum daily temperature above a threshold (default: 0℃) and the first occurrence of at least N (def 5) consecutive days with minimum daily temperature below the same threshold A mid date can be given to limit the earliest day the end of season can take. WARNING: The default freq and mid_date values are valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice `frost_free_season_length()`.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold to mark the beginning and end of frost free season. Default : 5.
- **mid_date** (*date (string, MM-DD)*) – Date the must be included in the season. It is the earliest the end of the season can be. If None, there is no limit. Default : 07-01.
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

frost_free_season_length (*DataArray*) – Length of the frost free season (days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days between the first occurrence of at least {window} consecutive days with minimum daily temperature above or at the freezing point and the first occurrence of at least {window} consecutive days with minimum daily temperature below freezing after {mid_date}.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then counted is the number of days between the first occurrence of at least N consecutive days with:

$$TN_{ij} >= 0$$

and the first subsequent occurrence of at least N consecutive days with:

$$TN_{ij} < 0$$

`xclim.indicators.atmos._temperature.frost_free_season_start(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '0 degC', window: int = 5, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Start of the frost free season. (realm: atmos)

Day of the year of the start of a sequence of days with minimum temperatures consistently above or equal to a threshold, after a period with minimum temperatures consistently above the same threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *frost_free_season_start()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold needed for evaluation. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

frost_free_season_start (*DataArray*) – Day of year of frost free season start (day_of_year) description: Day of year of beginning of frost free season, defined as the first day a minimum temperature threshold of {thresh} is equal or exceeded for at least {window} days.

Notes

Let x_i be the daily mean temperature at day of the year i for values of i going from 1 to 365 or 366. The start date of the start of growing season is given by the smallest index i for which:

$$\prod_{j=i}^{i+w} [x_j >= thresh]$$

is true, where w is the number of days the temperature threshold should be met or exceeded, and $[P]$ is 1 if P is true, and 0 if false.

`xclim.indicators.atmos._temperature.frost_season_length`(*tasmin: Union[DataArray, str] = 'tasmin', *, window: int = 5, mid_date: DayOfYearStr | None = '01-01', freq: str = 'AS-JUL', ds: Dataset = None*) → DataArray

Frost season length. (realm: atmos)

The number of days between the first occurrence of at least N (def: 5) consecutive days with minimum daily temperature under a threshold (default: 0℃) and the first occurrence of at least N (def 5) consecutive days with minimum daily temperature above the same threshold A mid date can be given to limit the earliest day the end of season can take. WARNING: The default freq and mid_date values are valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *frost_season_length()*. With injected parameters: thresh=0 degC.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature below threshold to mark the beginning and end of frost season. Default : 5.

- **mid_date** (*date (string, MM-DD)*) – Date the must be included in the season. It is the earliest the end of the season can be. If None, there is no limit. Default : 01-01.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

frost_season_length (*DataArray*) – Length of the frost season (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days between the first occurrence of at least {window} consecutive days with minimum daily temperature below freezing and the first occurrence of at least {window} consecutive days with minimuim daily temperature above freezing after {mid_date}.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then counted is the number of days between the first occurrence of at least N consecutive days with:

$$TN_{ij} > 0$$

and the first subsequent occurrence of at least N consecutive days with:

$$TN_{ij} < 0$$

`xclim.indicators.atmos._temperature.growing_degree_days(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '4.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Growing degree-days over threshold temperature value. (realm: atmos)

The sum of degree-days over the threshold temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *growing_degree_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 4.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

growing_degree_days (*DataArray*) – Growing degree days above {thresh} (integral_of_air_temperature_excess_wrt_time) [K days] cell_methods: time: sum over days description: {freq} growing degree days above {thresh}.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then the growing degree days are:

$$GD4_j = \sum_{i=1}^{I}(TG_{ij} - 4|TG_{ij} > 4)$$

`xclim.indicators.atmos._temperature.growing_season_end(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '5.0 degC', mid_date: DayOfYearStr = '07-01', window: int = 5, freq: str = 'YS', ds: Dataset = None*) → DataArray

End of the growing season. (realm: atmos)

Day of the year of the start of a sequence of days with mean temperatures consistently below a threshold, after a period with mean temperatures consistently above the same threshold.

This indicator will check for missing values according to the method "from_context". Based on indice `growing_season_end()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 5.0 degC. [Required units : [temperature]]
- **mid_date** (*date (string, MM-DD)*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'. Default : 07-01.
- **window** (*number*) – Minimum number of days with temperature below threshold needed for evaluation. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

growing_season_end (*DataArray*) – Day of year of growing season end (day_of_year) description: Day of year of end of growing season, defined as the first day of consistent inferior threshold temperature of {thresh} after a run of {window} days superior to threshold temperature.

`xclim.indicators.atmos._temperature.growing_season_length(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '5.0 degC', window: int = 6, mid_date: DayOfYearStr = '07-01', freq: str = 'YS', ds: Dataset = None*) → DataArray

Growing season length. (realm: atmos)

The number of days between the first occurrence of at least six consecutive days with mean daily temperature over a threshold (default: 5℃) and the first occurrence of at least six consecutive days with mean daily temperature below the same threshold after a certain date. (Usually July 1st in the northern emisphere and January 1st in the southern hemisphere.)

This indicator will check for missing values according to the method "from_context". Based on indice `growing_season_length()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 5.0 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold to mark the beginning and end of growing season. Default : 6.
- **mid_date** (*date (string, MM-DD)*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'. Default : 07-01.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

growing_season_length (*DataArray*) – ETCCDI Growing Season Length (Tmean > {thresh}) (growing_season_length) [days] description: {freq} number of days between the first occurrence of at least {window} consecutive days with mean daily temperature over {thresh} and the first occurrence of at least {window} consecutive days with mean daily temperature below {thresh} after {mid_date}.

Notes

Let TG_{ij} be the mean temperature at day i of period j. Then counted is the number of days between the first occurrence of at least 6 consecutive days with:

$$TG_{ij} > 5$$

and the first occurrence after 1 July of at least 6 consecutive days with:

$$TG_{ij} < 5$$

`xclim.indicators.atmos._temperature.growing_season_start(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '5.0 degC', window: int = 5, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Start of the growing season. (realm: atmos)

Day of the year of the start of a sequence of days with mean temperatures consistently above or equal to a threshold, after a period with mean temperatures consistently above the same threshold.

This indicator will check for missing values according to the method "from_context". Based on indice `growing_season_start()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 5.0 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold needed for evaluation. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

growing_season_start (*DataArray*) – Day of year of growing season start (day_of_year) description: Day of year of start of growing season, defined as the first day of consistent superior or equal to threshold temperature of {thresh} after a run of {window} days inferior to threshold temperature.

Notes

Let x_i be the daily mean temperature at day of the year i for values of i going from 1 to 365 or 366. The start date of the start of growing season is given by the smallest index i for which:

$$\prod_{j=i}^{i+w} [x_j >= thresh]$$

is true, where w is the number of days the temperature threshold should be met or exceeded, and $[P]$ is 1 if P is true, and 0 if false.

`xclim.indicators.atmos._temperature.heat_wave_frequency`(*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '22.0 degC', thresh_tasmax: str = '30 degC', window: int = 3, freq: str = 'YS', ds: Dataset = None*) → DataArray

Heat wave frequency. (realm: atmos)

Number of heat waves over a given period. A heat wave is defined as an event where the minimum and maximum daily temperature both exceeds specific thresholds over a minimum number of days.

This indicator will check for missing values according to the method "from_context". Based on indice *heat_wave_frequency()*. Keywords : health,.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The minimum temperature threshold needed to trigger a heatwave event. Default : 22.0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The maximum temperature threshold needed to trigger a heatwave event. Default : 30 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

heat_wave_frequency (*DataArray*) – Number of heat wave events (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax} for >= {window} days) (heat_wave_events) description: {freq} number of heat wave events over a given period. An event occurs when the minimum and maximum daily temperature both exceeds specific thresholds : (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax}) over a minimum number of days ({window}).

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities ([casati2013]).

In Robinson (2001; [robinson2001]), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

`xclim.indicators.atmos._temperature.heat_wave_index`(*tasmax: Union[DataArray, str] = 'tasmax', *, thresh: str = '25.0 degC', window: int = 5, freq: str = 'YS', ds: Dataset = None*) → DataArray

Heat wave index. (realm: atmos)

Number of days that are part of a heatwave, defined as five or more consecutive days over 25℃.

This indicator will check for missing values according to the method "from_context". Based on indice *heat_wave_index()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to designate a heatwave. Default : 25.0 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold to qualify as a heatwave. Default : 5.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

heat_wave_index (*DataArray*) – Number of days that are part of a heatwave (heat_wave_index) [days] description: {freq} number of days that are part of a heatwave, defined as five or more consecutive days over {thresh}.

`xclim.indicators.atmos._temperature.heat_wave_max_length`(*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '22.0 degC', thresh_tasmax: str = '30 degC', window: int = 3, freq: str = 'YS', ds: Dataset = None*) → DataArray

Heat wave max length. (realm: atmos)

Maximum length of heat waves over a given period. A heat wave is defined as an event where the minimum and maximum daily temperature both exceeds specific thresholds over a minimum number of days.

This indicator will check for missing values according to the method "from_context". Based on indice `heat_wave_max_length()`. Keywords : health,.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The minimum temperature threshold needed to trigger a heatwave event. Default : 22.0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The maximum temperature threshold needed to trigger a heatwave event. Default : 30 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

heat_wave_max_length (*DataArray*) – Maximum length of heat wave events (Tmin > {thresh_tasmin}and Tmax > {thresh_tasmax} for >= {window} days) (spell_length_of_days_with_air_temperature_above_threshold) [days] description: {freq} maximum length of heat wave events occurring in a given period. An event occurs when the minimum and maximum daily temperature both exceeds specific thresholds (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax}) over a minimum number of days ({window}).

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities ([casati2013]).

In Robinson (2001; [robinson2001]), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

`xclim.indicators.atmos._temperature.heat_wave_total_length(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '22.0 degC', thresh_tasmax: str = '30 degC', window: int = 3, freq: str = 'YS', ds: Dataset = None*) → DataArray

Heat wave total length. (realm: atmos)

Total length of heat waves over a given period. A heat wave is defined as an event where the minimum and maximum daily temperature both exceeds specific thresholds over a minimum number of days. This the sum of all days in such events.

This indicator will check for missing values according to the method "from_context". Based on indice *heat_wave_total_length()*. Keywords : health,.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – The minimum temperature threshold needed to trigger a heatwave event. Default : 22.0 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The maximum temperature threshold needed to trigger a heatwave event. Default : 30 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

heat_wave_total_length (*DataArray*) – Total length of heat wave events (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax} for >= {window} days) (spell_length_of_days_with_air_temperature_above_threshold) [days] description: {freq} total length of heat wave events occurring in a given period. An event occurs when the minimum and maximum daily temperature both exceeds specific thresholds (Tmin > {thresh_tasmin} and Tmax > {thresh_tasmax}) over a minimum number of days ({window}).

Notes

See notes and references of *heat_wave_max_length*

`xclim.indicators.atmos._temperature.heating_degree_days(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '17.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Heating degree days. (realm: atmos)

Sum of degree days below the temperature threshold at which spaces are heated.

This indicator will check for missing values according to the method "from_context". Based on indice *heating_degree_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 17.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

heating_degree_days (*DataArray*) – Heating degree days (Tmean < {thresh}) (integral_of_air_temperature_deficit_wrt_time) [K days] cell_methods: time: sum over days description: {freq} heating degree days below {thresh}.

Notes

This index intentionally differs from its ECA&D equivalent: HD17. In HD17, values below zero are not clipped before the sum. The present definition should provide a better representation of the energy demand for heating buildings to the given threshold.

Let TG_{ij} be the daily mean temperature at day i of period j. Then the heating degree days are:

$$HD17_j = \sum_{i=1}^{I} (17 - TG_{ij}) | TG_{ij} < 17)$$

`xclim.indicators.atmos._temperature.hot_spell_frequency(`*tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmax: str = '30 degC', window: int = 3, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Hot spell frequency. (realm: atmos)

Number of hot spells over a given period. A hot spell is defined as an event where the maximum daily temperature exceeds a specific threshold over a minimum number of days.

This indicator will check for missing values according to the method "from_context". Based on indice *hot_spell_frequency()*. Keywords : health,.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – The maximum temperature threshold needed to trigger a heatwave event. Default : 30 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

hot_spell_frequency (*DataArray*) – Number of hot spell events (Tmax > {thresh_tasmax} for >= {window} days) (hot_spell_events) description: {freq} number of hot spell events over a given period. An event occurs when the maximum daily temperature exceeds a specific threshold: (Tmax > {thresh_tasmax}) over a minimum number of days ({window}).

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities (Casati et al., 2013).

In Robinson (2001), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

Casati, B., A. Yagouti, and D. Chaumont, 2013: Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. J. Appl. Meteor. Climatol., 52, 2669–2698, https://doi.org/10.1175/JAMC-D-12-0341.1

Robinson, P.J., 2001: On the Definition of a Heat Wave. J. Appl. Meteor., 40, 762–775, https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

`xclim.indicators.atmos._temperature.hot_spell_max_length`(*tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmax: str = '30 degC', window: int = 1, freq: str = 'YS', ds: Dataset = None*) → DataArray

Longest hot spell. (realm: atmos)

Longest spell of high temperatures over a given period.

This indicator will check for missing values according to the method "from_context". Based on indice `hot_spell_max_length()`. Keywords : health,.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]

- **thresh_tasmax** (*quantity (string with units)*) – The maximum temperature threshold needed to trigger a heatwave event. Default : 30 degC. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

hot_spell_max_length (*DataArray*) – Maximum length of hot spell events (Tmax > {thresh_tasmax} for >= {window} days) (spell_length_of_days_with_air_temperature_above_threshold) [days] description: {freq} maximum length of hot spell events occurring in a given period. An event occurs when the maximum daily temperature exceeds a specific threshold: (Tmax > {thresh_tasmax}) over a minimum number of days ({window}).

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities (Casati et al., 2013).

In Robinson (2001), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

Casati, B., A. Yagouti, and D. Chaumont, 2013: Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. J. Appl. Meteor. Climatol., 52, 2669–2698, https://doi.org/10.1175/JAMC-D-12-0341.1

Robinson, P.J., 2001: On the Definition of a Heat Wave. J. Appl. Meteor., 40, 762–775, https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

`xclim.indicators.atmos._temperature.huglin_index(`*tas: Union[DataArray, str] = 'tas', tasmax: Union[DataArray, str] = 'tasmax', lat: Union[DataArray, str] = 'lat', *, thresh: str = '10 degC', start_date: DayOfYearStr = '04-01', end_date: DayOfYearStr = '10-01', freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Huglin Heliothermal Index. (realm: atmos)

Growing-degree days with a base of 10°C and adjusted for latitudes between 40°N and 50°N for April to September (Northern Hemisphere; October to March in Southern Hemisphere). Originally proposed in [Huglin1978]. Used as a heat-summation metric in viticulture agroclimatology.

This indicator will check for missing values according to the method "from_context". Based on indice `huglin_index()`. With injected parameters: method=jones.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **lat** (*str or DataArray*) – Latitude coordinate. Default : *ds.lat*. [Required units : []]
- **thresh** (*quantity (string with units)*) – The temperature threshold. Default : 10 degC. [Required units : [temperature]]
- **start_date** (*date (string, MM-DD)*) – The hemisphere-based start date to consider (north = April, south = October). Default : 04-01.
- **end_date** (*date (string, MM-DD)*) – The hemisphere-based start date to consider (north = October, south = April). This date is non-inclusive. Default : 10-01.
- **freq** (*offset alias (string)*) – Resampling frequency (default: "YS"; For Southern Hemisphere, should be "AS-JUL"). Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

hi (*DataArray*) – Huglin heliothermal index (Summation of ((Tmin + Tmax)/2 - {thresh}) * Latitude-based day-lengthcoefficient (*k*), for days between {start_date} and {end_date}). description: Heat-summation index for agroclimatic suitability estimation, developed specifically for viticulture. Considers daily Tmin and Tmax with a base of {thresh}, typically between 1 April and 30 September. Integrates a day-length coefficient calculation for higher latitudes. comment: Metric originally published in Huglin (1978). Day-length coefficient based on Hall & Jones (2010)

Notes

Let TX_i and TG_i be the daily maximum and mean temperature at day i and T_{thresh} the base threshold needed for heat summation (typically, 10 degC). A day-length multiplication, k, based on latitude, lat, is also considered. Then the Huglin heliothermal index for dates between 1 April and 30 September is:

$$HI = \sum_{i=\text{April 1}}^{\text{September 30}} \left(\frac{TX_i + TG_i)}{2} - T_{thresh} \right) * k$$

For the *smoothed* method, the day-length multiplication factor, k, is calculated as follows:

$$k = f(lat) = \begin{cases} 1, & \text{if } |lat| <= 40 \\ 1 + ((abs(lat) - 40)/10) * 0.06, & \text{if } 40 < |lat| <= 50 \\ NaN, & \text{if } |lat| > 50 \end{cases}$$

For compatibility with ICCLIM, *end_ date* should be set to *11-01*, *method* should be set to *icclim*. The day-length multiplication factor, k, is calculated as follows:

$$k = f(lat) = \begin{cases} 1.0, & \text{if } |lat| <= 40 \\ 1.02, & \text{if } 40 < |lat| <= 42 \\ 1.03, & \text{if } 42 < |lat| <= 44 \\ 1.04, & \text{if } 44 < |lat| <= 46 \\ 1.05, & \text{if } 46 < |lat| <= 48 \\ 1.06, & \text{if } 48 < |lat| <= 50 \\ NaN, & \text{if } |lat| > 50 \end{cases}$$

A more robust day-length calculation based on latitude, calendar, day-of-year, and obliquity is available with *method="jones"*. See: `xclim.indices.generic.day_lengths()` or [Hall&Jones2010]_ for more information.

References

`xclim.indicators.atmos._temperature.ice_days`(*tasmax: Union[DataArray, str] = 'tasmax'*, ***, *thresh: str = '0 degC'*, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of ice/freezing days. (realm: atmos)

Number of days where daily maximum temperatures are below a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *ice_days()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Freezing temperature. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

ice_days (*DataArray*) – Number of ice days (Tmax < {thresh}) (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with maximum daily temperature below {thresh}.

Notes

Let TX_{ij} be the daily maximum temperature at day i of period j, and :math`TT` the threshold. Then counted is the number of days where:

$$TX_{ij} < TT$$

`xclim.indicators.atmos._temperature.last_spring_frost`(*tas: Union[DataArray, str] = 'tas'*, ***, *thresh: str = '0 degC'*, *before_date: DayOfYearStr = '07-01'*, *window: int = 1*, *freq: str = 'YS'*, *ds: Dataset = None*) → DataArray

Last day of temperatures inferior to a threshold temperature. (realm: atmos)

Returns last day of period where a temperature is inferior to a threshold over a given number of days and limited to a final calendar date.

This indicator will check for missing values according to the method "from_context". Based on indice *last_spring_frost()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]

- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **before_date** (*date (string, MM-DD)*) – Date of the year before which to look for the final frost event. Should have the format '%m-%d'. Default : 07-01.
- **window** (*number*) – Minimum number of days with temperature below threshold needed for evaluation. Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
last_spring_frost (*DataArray*) – Day of year of last spring frost (day_of_year) description: Day of year of last spring frost, defined as the last day a minimum temperature threshold of {thresh} is not exceeded before a given date.

`xclim.indicators.atmos._temperature.latitude_temperature_index(`*tas: Union[DataArray, str] = 'tas', lat: Union[DataArray, str] = 'lat', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Latitude-Temperature Index. (realm: atmos)

Mean temperature of the warmest month with a latitude-based scaling factor ([Jackson&Cherry1988]_). Used for categorizing wine-growing regions.

This indicator will check for missing values according to the method "from_context". Based on indice *`latitude_temperature_index()`*. With injected parameters: lat_factor=60.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **lat** (*str or DataArray*) – Latitude coordinate. Default : *ds.lat*. [Required units : []]
- **freq** (*offset alias (string)*) – Resampling frequency. Restricted to frequencies equivalent to one of ['A'] Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
lti (*DataArray*) – Latitude-temperature index description: A climate indice based on mean temperature of the warmest month and a latitude-based coefficient to account for longer day-length favouring growing conditions. Developed specifically for viticulture. Mean temperature of warmest month * ({lat_factor} - latitude). comment: Indice originally published in Jackson, D. I., & Cherry, N. J. (1988)

Notes

The latitude factor of *75* is provided for examining the poleward expansion of wine-growing climates under scenarios of climate change (modified from [Kenny&Shao1992]_). For comparing 20th century/observed historical records, the original scale factor of *60* is more appropriate.

Let Tn_j be the average temperature for a given month j, lat_f be the latitude factor, and lat be the latitude of the area of interest. Then the Latitude-Temperature Index (LTI) is:

$$LTI = max(TN_j : j = 1..12)(lat_f - |lat|)$$

References

`xclim.indicators.atmos._temperature.max_daily_temperature_range(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Maximum of daily temperature range. (realm: atmos)

The mean difference between the daily maximum temperature and the daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *daily_temperature_range()*. With injected parameters: op=max.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

dtrmax (*DataArray*) – Maximum Diurnal Temperature Range (air_temperature) [K] cell_methods: time range within days time: max over days description: {freq} maximum diurnal temperature range.

Notes

For a default calculation using *op='mean'* :

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the mean diurnal temperature range in period j is:

$$DTR_j = \frac{\sum_{i=1}^{I}(TX_{ij} - TN_{ij})}{I}$$

`xclim.indicators.atmos._temperature.maximum_consecutive_frost_free_days(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Maximum number of consecutive frost free days (Tn >= 0°C). (realm: atmos)

Return the maximum number of consecutive days within the period where the minimum temperature is above or equal to a certain threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *maximum_consecutive_frost_free_days()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

consecutive_frost_free_days (*DataArray*) – Maximum number of consecutive days with Tmin >= {thresh} (spell_length_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: maximum over days description: {freq} maximum number of consecutive days with minimum daily temperature above or equal to {thresh}.

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily minimum temperature series and *thresh* the threshold above or equal to which a day is considered a frost free day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i <= thresh] \neq [t_{i+1} <= thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive frost free days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} >= thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indicators.atmos._temperature.maximum_consecutive_warm_days(`*tasmax: Union[DataArray, str] = 'tasmax', *, thresh: str = '25 degC', freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Maximum number of consecutive days with tasmax above a threshold (summer days). (realm: atmos)

Return the maximum number of consecutive days within the period where the maximum temperature is above a certain threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *maximum_consecutive_tx_days()*.

Parameters

- **tasmax** (*str or DataArray*) – Max daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature. Default : 25 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

maximum_consecutive_warm_days (*DataArray*) – The maximum number of days with tasmax > thresh per periods (summer days). (spell_length_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: maximum over days description: {freq} longest spell of consecutive days with Tmax above {thresh}.

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily maximum temperature series and *thresh* the threshold above which a day is considered a summer day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i < thresh] \neq [t_{i+1} < thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive dry days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

```
xclim.indicators.atmos._temperature.tg10p(tas: Union[DataArray, str] = 'tas', tas_per: Union[DataArray, str] = 'tas_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer) → DataArray
```

Number of days with daily mean temperature below the 10th percentile. (realm: atmos)

Number of days with daily mean temperature below the 10th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tg10p()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **tas_per** (*str or DataArray*) – 10th percentile of daily mean temperature. Default : *ds.tas_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tg10p (*DataArray*) – Number of days when Tmean < {tas_per_thresh}th percentile (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with mean daily temperature below the {tas_per_thresh}th percentile(s). A {tas_per_window} day(s) window, centred on each calendar day in the {tas_per_period} period, is used to compute the {tas_per_thresh}th percentile(s).

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

`xclim.indicators.atmos._temperature.tg90p(`*tas: Union[DataArray, str] = 'tas', tas_per: Union[DataArray, str] = 'tas_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily mean temperature over the 90th percentile. (realm: atmos)

Number of days with daily mean temperature over the 90th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *`tg90p()`*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **tas_per** (*str or DataArray*) – 90th percentile of daily mean temperature. Default : *ds.tas_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tg90p (*DataArray*) – Number of days when Tmean > {tas_per_thresh}th percentile (days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with mean daily temperature above the the {tas_per_thresh}th percentile(s). A {tas_per_window} day(s) window, centred on each calendar day in the {tas_per_period} period, is used to compute the {tas_per_thresh}th percentile(s).

Notes

The 90th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

`xclim.indicators.atmos._temperature.tg_days_above(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '10.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with tas above a threshold. (realm: atmos)

Number of days where daily mean temperature exceeds a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tg_days_above()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 10.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tg_days_above (*DataArray*) – Number of days with Tavg > {thresh} (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days where daily mean temperature exceeds {thresh}.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then counted is the number of days where:

$$TG_{ij} > Threshold[]$$

`xclim.indicators.atmos._temperature.tg_days_below(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '10.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with tas below a threshold. (realm: atmos)

Number of days where daily mean temperature is below a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tg_days_below()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]

- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 10.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tg_days_below (*DataArray*) – Number of days with Tavg < {thresh} (number_of_days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days where daily mean temperature is below {thresh}.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then counted is the number of days where:

$$TG_{ij} < Threshold[]$$

`xclim.indicators.atmos._temperature.tg_max(`*tas: Union[DataArray, str] = 'tas', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Highest mean temperature. (realm: atmos)

The maximum of daily mean temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tg_max()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tg_max (*DataArray*) – Maximum daily mean temperature (air_temperature) [K] cell_methods: time: maximum over days description: {freq} maximum of daily mean temperature.

Notes

Let TN_{ij} be the mean temperature at day i of period j. Then the maximum daily mean temperature for period j is:

$$TNx_j = max(TN_{ij})$$

`xclim.indicators.atmos._temperature.tg_mean`(*tas: Union[DataArray, str] = 'tas'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Mean of daily average temperature. (realm: atmos)

Resample the original daily mean temperature series by taking the mean over each period.

This indicator will check for missing values according to the method "from_context". Based on indice `tg_mean()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tg_mean (*DataArray*) – Mean daily mean temperature (air_temperature) [K] cell_methods: time: mean over days description: {freq} mean of daily mean temperature.

Notes

Let TN_i be the mean daily temperature of day i, then for a period p starting at day a and finishing on day b:

$$TG_p = \frac{\sum_{i=a}^{b} TN_i}{b - a + 1}$$

`xclim.indicators.atmos._temperature.tg_min`(*tas: Union[DataArray, str] = 'tas'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Lowest mean temperature. (realm: atmos)

Minimum of daily mean temperature.

This indicator will check for missing values according to the method "from_context". Based on indice `tg_min()`.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
tg_min (*DataArray*) – Minimum daily mean temperature (air_temperature) [K] cell_methods: time: minimum over days description: {freq} minimum of daily mean temperature.

Notes

Let TG_{ij} be the mean temperature at day i of period j. Then the minimum daily mean temperature for period j is:

$$TGn_j = min(TG_{ij})$$

`xclim.indicators.atmos._temperature.thawing_degree_days(`*tas: Union[DataArray, str] = 'tas', *, thresh: str = '0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Growing degree-days over threshold temperature value. (realm: atmos)

The sum of degree-days over the threshold temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *growing_degree_days()*.

Parameters

- **tas** (*str or DataArray*) – Mean daily temperature. Default : *ds.tas*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
thawing_degree_days (*DataArray*) – Thawing degree days (degree days above 0°C) (integral_of_air_temperature_excess_wrt_time) [K days] cell_methods: time: sum over days description: {freq} thawing degree days above 0°C.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then the growing degree days are:

$$GD4_j = \sum_{i=1}^{I} (TG_{ij} - 4 | TG_{ij} > 4)$$

`xclim.indicators.atmos._temperature.tn10p(`*tasmin: Union[DataArray, str] = 'tasmin', tasmin_per: Union[DataArray, str] = 'tasmin_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily minimum temperature below the 10th percentile. (realm: atmos)

Number of days with daily minimum temperature below the 10th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tn10p()*.

Parameters

- **tasmin** (*str or DataArray*) – Mean daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmin_per** (*str or DataArray*) – 10th percentile of daily minimum temperature. Default : *ds.tasmin_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn10p (*DataArray*) – Number of days when Tmin < {tasmin_per_thresh}th percentile (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with minimum daily temperature below the the {tasmin_per_thresh}th percentile(s). A {tasmin_per_window} day(s) window, centred on each calendar day in the {tasmin_per_period} period, is used to compute the {tasmin_per_thresh}th percentile(s).

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

`xclim.indicators.atmos._temperature.tn90p(`*tasmin: Union[DataArray, str] = 'tasmin', tasmin_per: Union[DataArray, str] = 'tasmin_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily minimum temperature over the 90th percentile. (realm: atmos)

Number of days with daily minimum temperature over the 90th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tn90p()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmin_per** (*str or DataArray*) – 90th percentile of daily minimum temperature. Default : *ds.tasmin_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn90p (*DataArray*) – Number of days when Tmin > {tasmin_per_thresh}th percentile (days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with minimum daily temperature above the the {tasmin_per_thresh}th percentile(s). A {tasmin_per_window} day(s) window, centred on each calendar day in the {tasmin_per_period} period, is used to compute the {tasmin_per_thresh}th percentile(s).

Notes

The 90th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

`xclim.indicators.atmos._temperature.tn_days_above(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '20.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with tasmin above a threshold (number of tropical nights). (realm: atmos)

Number of days where daily minimum temperature exceeds a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_days_above()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 20.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn_days_above (*DataArray*) – Number of days with Tmin > {thresh} (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days where daily minimum temperature exceeds {thresh}.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} > Threshold[]$$

`xclim.indicators.atmos._temperature.tn_days_below(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '-10.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with tasmin below a threshold. (realm: atmos)

Number of days where daily minimum temperature is below a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_days_below()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : -10.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn_days_below (*DataArray*) – Number of days with Tmin < {thresh} (number_of_days_with_air_temperature_below_threshold) [days] cell_methods: time:

sum over days description: {freq} number of days where daily minimum temperature is below {thresh}.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} < Threshold[]$$

`xclim.indicators.atmos._temperature.tn_max(`*tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Highest minimum temperature. (realm: atmos)

The maximum of daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_max()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tn_max (*DataArray*) – Maximum daily minimum temperature (air_temperature) [K] cell_methods: time: maximum over days description: {freq} maximum of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then the maximum daily minimum temperature for period j is:

$$TNx_j = max(TN_{ij})$$

`xclim.indicators.atmos._temperature.tn_mean(`*tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Mean minimum temperature. (realm: atmos)

Mean of daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_mean()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]

- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
tn_mean (*DataArray*) – Mean daily minimum temperature (air_temperature) [K] cell_methods: time: mean over days description: {freq} mean of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then mean values in period j are given by:

$$TN_{ij} = \frac{\sum_{i=1}^{I} TN_{ij}}{I}$$

`xclim.indicators.atmos._temperature.tn_min(`*tasmin: Union[DataArray, str] = 'tasmin', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Lowest minimum temperature. (realm: atmos)

Minimum of daily minimum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_min()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
tn_min (*DataArray*) – Minimum daily minimum temperature (air_temperature) [K] cell_methods: time: minimum over days description: {freq} minimum of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then the minimum daily minimum temperature for period j is:

$$TNn_j = min(TN_{ij})$$

`xclim.indicators.atmos._temperature.tropical_nights(`*tasmin: Union[DataArray, str] = 'tasmin', *, thresh: str = '20.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with tasmin above a threshold (number of tropical nights). (realm: atmos)

Number of days where daily minimum temperature exceeds a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tn_days_above()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 20.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tropical_nights (*DataArray*) – Number of Tropical Nights (Tmin > {thresh}) (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of Tropical Nights : defined as days with minimum daily temperature above {thresh}.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} > Threshold[]$$

`xclim.indicators.atmos._temperature.tx10p(`*tasmax: Union[DataArray, str] = 'tasmax', tasmax_per: Union[DataArray, str] = 'tasmax_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily maximum temperature below the 10th percentile. (realm: atmos)

Number of days with daily maximum temperature below the 10th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *tx10p()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **tasmax_per** (*str or DataArray*) – 10th percentile of daily maximum temperature. Default : *ds.tasmax_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.

- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx10p (*DataArray*) – Number of days when Tmax < {tasmax_per_thresh}th percentile (days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with maximum daily temperature below the {tasmax_per_thresh}th percentile(s). A {tasmax_per_window} day(s) window, centred on each calendar day in the {tasmax_per_period} period, is used to compute the {tasmax_per_thresh}th percentile(s).

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

`xclim.indicators.atmos._temperature.tx90p(`*tasmax: Union[DataArray, str] = 'tasmax', tasmax_per: Union[DataArray, str] = 'tasmax_per', *, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None, **indexer*`)` → DataArray

Number of days with daily maximum temperature over the 90th percentile. (realm: atmos)

Number of days with daily maximum temperature over the 90th percentile.

This indicator will check for missing values according to the method "from_context". Based on indice *`tx90p()`*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **tasmax_per** (*str or DataArray*) – 90th percentile of daily maximum temperature. Default : *ds.tasmax_per*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx90p (*DataArray*) – Number of days when Tmax > {tasmax_per_thresh}th percentile (days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with maximum daily temperature above the {tasmax_per_thresh}th percentile(s). A {tasmax_per_window} day(s) window, centred on each calendar day in the {tasmax_per_period} period, is used to compute the {tasmax_per_thresh}th percentile(s).

Notes

The 90th percentile should be computed for a 5-day window centered on each calendar day for a reference period.

`xclim.indicators.atmos._temperature.tx_days_above(`*tasmax: Union[DataArray, str] = 'tasmax', *, thresh: str = '25.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with tasmax above a threshold (number of summer days). (realm: atmos)

Number of days where daily maximum temperature exceeds a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_days_above()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 25.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx_days_above (*DataArray*) – Number of days with Tmax > {thresh} (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days where daily maximum temperature exceeds {thresh}.

Notes

Let TX_{ij} be the daily maximum temperature at day i of period j. Then counted is the number of days where:

$$TX_{ij} > Threshold[]$$

`xclim.indicators.atmos._temperature.tx_days_below(`*tasmax: Union[DataArray, str] = 'tasmax', *, thresh: str = '25.0 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with tmax below a threshold. (realm: atmos)

Number of days where daily maximum temperature is below a threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_days_below()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh** (*quantity (string with units)*) – Threshold temperature on which to base evaluation. Default : 25.0 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx_days_below (*DataArray*) – Number of days with Tmax < {thresh} (number_of_days_with_air_temperature_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days where daily max temperature is below {thresh}.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} < Threshold[]$$

`xclim.indicators.atmos._temperature.tx_max(`*tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Highest max temperature. (realm: atmos)

The maximum value of daily maximum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_max()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
tx_max (*DataArray*) – Maximum daily maximum temperature (air_temperature) [K] cell_methods: time: maximum over days description: {freq} maximum of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then the maximum daily maximum temperature for period j is:

$$TXx_j = max(TX_{ij})$$

`xclim.indicators.atmos._temperature.tx_mean`(*tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Mean max temperature. (realm: atmos)

The mean of daily maximum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_mean()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
tx_mean (*DataArray*) – Mean daily maximum temperature (air_temperature) [K] cell_methods: time: mean over days description: {freq} mean of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then mean values in period j are given by:

$$TX_{ij} = \frac{\sum_{i=1}^{I} TX_{ij}}{I}$$

`xclim.indicators.atmos._temperature.tx_min`(*tasmax: Union[DataArray, str] = 'tasmax', *, freq: str = 'YS', ds: Dataset = None, **indexer*) → DataArray

Lowest max temperature. (realm: atmos)

The minimum of daily maximum temperature.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_min()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx_min (*DataArray*) – Minimum daily maximum temperature (air_temperature) [K] cell_methods: time: minimum over days description: {freq} minimum of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then the minimum daily maximum temperature for period j is:

$$TXn_j = min(TX_{ij})$$

`xclim.indicators.atmos._temperature.tx_tn_days_above(`*tasmin: Union[DataArray, str] = 'tasmin', tasmax: Union[DataArray, str] = 'tasmax', *, thresh_tasmin: str = '22 degC', thresh_tasmax: str = '30 degC', freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Number of days with both hot maximum and minimum daily temperatures. (realm: atmos)

The number of days per period with tasmin above a threshold and tasmax above another threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *tx_tn_days_above()*.

Parameters

- **tasmin** (*str or DataArray*) – Minimum daily temperature. Default : *ds.tasmin*. [Required units : [temperature]]
- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **thresh_tasmin** (*quantity (string with units)*) – Threshold temperature for tasmin on which to base evaluation. Default : 22 degC. [Required units : [temperature]]
- **thresh_tasmax** (*quantity (string with units)*) – Threshold temperature for tasmax on which to base evaluation. Default : 30 degC. [Required units : [temperature]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

tx_tn_days_above (*DataArray*) – Number of days with Tmax > {thresh_tasmax} and Tmin > {thresh_tasmin} (number_of_days_with_air_temperature_above_threshold) [days] description: {freq} number of days where daily maximum temperature exceeds {thresh_tasmax} and minimum temperature exceeds {thresh_tasmin}.

Notes

Let TX_{ij} be the maximum temperature at day i of period j, TN_{ij} the daily minimum temperature at day i of period j, TX_{thresh} the threshold for maximum daily temperature, and TN_{thresh} the threshold for minimum daily temperature. Then counted is the number of days where:

$$TX_{ij} > TX_{thresh}[]$$

and where:

$$TN_{ij} > TN_{thresh}[]$$

`xclim.indicators.atmos._temperature.warm_spell_duration_index(`*tasmax: Union[DataArray, str] = 'tasmax', tasmax_per: Union[DataArray, str] = 'tasmax_per', *, window: int = 6, freq: str = 'YS', bootstrap: bool = False, ds: Dataset = None*) → DataArray

Warm spell duration index. (realm: atmos)

Number of days inside spells of a minimum number of consecutive days where the daily maximum temperature is above the 90th percentile. The 90th percentile should be computed for a 5-day moving window, centered on each calendar day in the 1961-1990 period.

This indicator will check for missing values according to the method "from_context". Based on indice *warm_spell_duration_index()*.

Parameters

- **tasmax** (*str or DataArray*) – Maximum daily temperature. Default : *ds.tasmax*. [Required units : [temperature]]
- **tasmax_per** (*str or DataArray*) – percentile(s) of daily maximum temperature. Default : *ds.tasmax_per*. [Required units : [temperature]]
- **window** (*number*) – Minimum number of days with temperature above threshold to qualify as a warm spell. Default : 6.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **bootstrap** (*boolean*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive. Default : False.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

warm_spell_duration_index (*DataArray*) – Number of days part of a percentile-defined warm spell (number_of_days_with_air_temperature_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with at least {window} consecutive days where the daily maximum temperature is above the {tasmax_per_thresh}th percentile(s). A {tasmax_per_window} day(s) window, centred on each calendar day in the {tasmax_per_period} period, is used to compute the {tasmax_per_thresh}th percentile(s).

References

From the Expert Team on Climate Change Detection, Monitoring and Indices (ETCCDMI). Used in Alexander, L. V., et al. (2006), Global observed changes in daily climate extremes of temperature and precipitation, J. Geophys. Res., 111, D05109, doi: 10.1029/2005JD006290.

xclim.indicators.atmos._wind module

`xclim.indicators.atmos._wind.calm_days`(*sfcWind: Union[DataArray, str] = 'sfcWind'*, ***, *thresh: str = '2 m s-1'*, *freq: str = 'MS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Calm days. (realm: atmos)

The number of days with average near-surface wind speed below threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *calm_days()*.

Parameters

- **sfcWind** (*str or DataArray*) – Daily windspeed. Default : *ds.sfcWind*. [Required units : [speed]]
- **thresh** (*quantity (string with units)*) – Threshold average near-surface wind speed on which to base evaluation. Default : 2 m s-1. [Required units : [speed]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

calm_days (*DataArray*) – Number of days with surface wind speed below threshold (number_of_days_with_sfcWind_below_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with surface wind speed < {thresh}

Notes

Let WS_{ij} be the windspeed at day i of period j. Then counted is the number of days where:

$$WS_{ij} < Threshold[ms-1]$$

`xclim.indicators.atmos._wind.windy_days(`*sfcWind: Union[DataArray, str] = 'sfcWind', *, thresh: str = '10.8 m s-1', freq: str = 'MS', ds: Dataset = None, **indexer*`)` → DataArray

Windy days. (realm: atmos)

The number of days with average near-surface wind speed above threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *windy_days()*.

Parameters

- **sfcWind** (*str or DataArray*) – Daily average near-surface wind speed. Default : *ds.sfcWind*. [Required units : [speed]]
- **thresh** (*quantity (string with units)*) – Threshold average near-surface wind speed on which to base evaluation. Default : 10.8 m s-1. [Required units : [speed]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : MS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

windy_days (*DataArray*) – Number of days with surface wind speed above threshold (number_of_days_with_sfcWind_above_threshold) [days] cell_methods: time: sum over days description: {freq} number of days with surface wind speed >= {thresh}

Notes

Let WS_{ij} be the windspeed at day i of period j. Then counted is the number of days where:

$$WS_{ij} >= Threshold[ms-1]$$

xclim.indicators.land package

Land indicators

Submodules

xclim.indicators.land._snow module

`xclim.indicators.land._snow.blowing_snow(`*snd: Union[DataArray, str] = 'snd', sfcWind: Union[DataArray, str] = 'sfcWind', *, snd_thresh: str = '5 cm', sfcWind_thresh: str = '15 km/h', window: int = 3, freq: str = 'AS-JUL', ds: Dataset = None*`)` → DataArray

Days with blowing snow events. (realm: land)

Number of days where both snowfall over the last days and daily wind speeds are above respective thresholds.

This indicator will check for missing values according to the method "from_context". Based on indice *blowing_snow()*.

Parameters

- **snd** (*str or DataArray*) – Surface snow depth. Default : *ds.snd*. [Required units : [length]]
- **sfcWind** (*str or DataArray*) – Wind velocity Default : *ds.sfcWind*. [Required units : [speed]]
- **snd_thresh** (*quantity (string with units)*) – Threshold on net snowfall accumulation over the last *window* days. Default : 5 cm. [Required units : [length]]
- **sfcWind_thresh** (*quantity (string with units)*) – Wind speed threshold. Default : 15 km/h. [Required units : [speed]]
- **window** (*number*) – Period over which snow is accumulated before comparing against threshold. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

{freq}_blowing_snow (*DataArray*) – Number of days where snowfall and wind speeds are above respective thresholds. [days] description: {freq} number of days with snowfall over last {window} days above {snd_thresh} and wind speed above {sfcWind_thresh}.

`xclim.indicators.land._snow.continuous_snow_cover_end`(*snd: Union[DataArray, str] = 'snd'*, ***, *thresh: str = '2 cm'*, *window: int = 14*, *freq: str = 'AS-JUL'*, *ds: Dataset = None*) → DataArray

End date of continuous snow cover. (realm: land)

First day after the start of the continuous snow cover when snow depth is below *threshold* for at least *window* consecutive days. WARNING: The default *freq* is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *continuous_snow_cover_end()*.

Parameters

- **snd** (*str or DataArray*) – Surface snow thickness. Default : *ds.snd*. [Required units : [length]]
- **thresh** (*quantity (string with units)*) – Threshold snow thickness. Default : 2 cm. [Required units : [length]]
- **window** (*number*) – Minimum number of days with snow depth below threshold. Default : 14.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

continuous_snow_cover_end (*DataArray*) – End date of continuous snow cover (day_of_year) description: Day of year when snow depth is below {thresh} for {window} consecutive days.

References

Chaumont D., Mailhot A., Diaconescu E.P., Fournier É., Logan T. 2017: Élaboration du portrait bioclimatique futur du Nunavik – Tome II. [Rapport présenté au Ministère de la forêt, de la faune et des parcs], Ouranos.

`xclim.indicators.land._snow.continuous_snow_cover_start`(*snd: Union[DataArray, str] = 'snd', *, thresh: str = '2 cm', window: int = 14, freq: str = 'AS-JUL', ds: Dataset = None*) → DataArray

Start date of continuous snow cover. (realm: land)

Day of year when snow depth is above or equal *threshold* for at least *window* consecutive days. WARNING: The default *freq* is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice *continuous_snow_cover_start()*.

Parameters

- **snd** (*str or DataArray*) – Surface snow thickness. Default : *ds.snd*. [Required units : [length]]
- **thresh** (*quantity (string with units)*) – Threshold snow thickness. Default : 2 cm. [Required units : [length]]
- **window** (*number*) – Minimum number of days with snow depth above or equal to threshold. Default : 14.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

continuous_snow_cover_start (*DataArray*) – Start date of continuous snow cover (day_of_year) description: Day of year when snow depth is above or equal to {thresh} for {window} consecutive days.

References

Chaumont D., Mailhot A., Diaconescu E.P., Fournier É., Logan T. 2017: Élaboration du portrait bioclimatique futur du Nunavik – Tome II. [Rapport présenté au Ministère de la forêt, de la faune et des parcs], Ouranos.

`xclim.indicators.land._snow.snd_max_doy`(*snd: Union[DataArray, str] = 'snd', *, freq: str = 'AS-JUL', ds: Dataset = None, **indexer*) → DataArray

Maximum snow depth day of year. (realm: land)

Day of year when surface snow reaches its peak value. If snow depth is 0 over entire period, return NaN.

This indicator will check for missing values according to the method "from_context". Based on indice *snd_max_doy()*.

Parameters

- **snd** (*str or DataArray*) – Surface snow depth. Default : *ds.snd*. [Required units : [length]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

{freq}_snd_max_doy (*DataArray*) – Date when snow depth reaches its maximum value. (day_of_year) description: {freq} day of year when snow depth reaches its maximum value.

`xclim.indicators.land._snow.snow_cover_duration`(*snd: Union[DataArray, str] = 'snd'*, ***, *thresh: str = '2 cm'*, *freq: str = 'AS-JUL'*, *ds: Dataset = None*, ***indexer*) → DataArray

Number of days with snow depth above a threshold. (realm: land)

Number of days where surface snow depth is greater or equal to given threshold. WARNING: The default *freq* is valid for the northern hemisphere.

This indicator will check for missing values according to the method "from_context". Based on indice `snow_cover_duration()`.

Parameters

- **snd** (*str or DataArray*) – Surface snow thickness. Default : *ds.snd*. [Required units : [length]]
- **thresh** (*quantity (string with units)*) – Threshold snow thickness. Default : 2 cm. [Required units : [length]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

snow_cover_duration (*DataArray*) – Number of days with snow depth above threshold [days] description: {freq} number of days with snow depth greater or equal to {thresh}

`xclim.indicators.land._snow.snow_depth`(*snd: Union[DataArray, str] = 'snd'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Mean of daily average snow depth. (realm: land)

Resample the original daily mean snow depth series by taking the mean over each period.

This indicator will check for missing values according to the method "from_context". Based on indice `snow_depth()`.

Parameters

- **snd** (*str or DataArray*) – Default : *ds.snd*. [Required units : [length]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.

- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
snow_depth (*DataArray*) – Mean of daily snow depth (surface_snow_thickness) [cm] cell_methods: time: mean over days description: {freq} mean of daily mean snow depth.

`xclim.indicators.land._snow.snow_melt_we_max`(*snw: Union[DataArray, str] = 'snw'*, ***, *window: int = 3*, *freq: str = 'AS-JUL'*, *ds: Dataset = None*) → DataArray

Maximum snow melt. (realm: land)

The maximum snow melt over a given number of days expressed in snow water equivalent.

This indicator will check for missing values according to the method "from_context". Based on indice *snow_melt_we_max()*.

Parameters

- **snw** (*str or DataArray*) – Snow amount (mass per area). Default : *ds.snw*. [Required units : [mass]/[area]]
- **window** (*number*) – Number of days during which the melt is accumulated. Default : 3.
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns
{freq}_snow_melt_we_max (*DataArray*) – The maximum snow melt over a given number of days for each period. [mass/area]. (change_over_time_in_surface_snow_amount) [kg m-2] description: {freq} maximum negative change in melt amount over {window} days.

`xclim.indicators.land._snow.snw_max`(*snw: Union[DataArray, str] = 'snw'*, ***, *freq: str = 'AS-JUL'*, *ds: Dataset = None*, ***indexer*) → DataArray

Maximum snow amount. (realm: land)

The maximum daily snow amount.

This indicator will check for missing values according to the method "from_context". Based on indice *snw_max()*.

Parameters

- **snw** (*str or DataArray*) – Snow amount (mass per area). Default : *ds.snw*. [Required units : [mass]/[area]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns
{freq}_snw_max (*DataArray*) – Maximum daily snow amount (sur-

face_snow_amount) [kg m-2] description: {freq} day of year when snow amount on the surface reaches its maximum.

`xclim.indicators.land._snow.snw_max_doy`(*snw: Union[DataArray, str] = 'snw'*, ***, *freq: str = 'AS-JUL'*, *ds: Dataset = None*, ***indexer*) → DataArray

Maximum snow amount day of year. (realm: land)

Day of year when surface snow amount reaches its peak value. If snow amount is 0 over entire period, return NaN.

This indicator will check for missing values according to the method "from_context". Based on indice *snw_max_doy()*.

Parameters

- **snw** (*str or DataArray*) – Surface snow amount. Default : *ds.snw*. [Required units : [mass]/[area]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

{freq}_snw_max_doy (*DataArray*) – Day of year of maximum daily snow amount (day_of_year) description: {freq} maximum snow amount on the surface.

`xclim.indicators.land._snow.winter_storm`(*snd: Union[DataArray, str] = 'snd'*, ***, *thresh: str = '25 cm'*, *freq: str = 'AS-JUL'*, *ds: Dataset = None*, ***indexer*) → DataArray

Days with snowfall over threshold. (realm: land)

Number of days with snowfall accumulation greater or equal to threshold.

This indicator will check for missing values according to the method "from_context". Based on indice *winter_storm()*.

Parameters

- **snd** (*str or DataArray*) – Surface snow depth. Default : *ds.snd*. [Required units : [length]]
- **thresh** (*quantity (string with units)*) – Threshold on snowfall accumulation require to label an event a *winter storm*. Default : 25 cm. [Required units : [length]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : AS-JUL.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Default : None.

Returns

{freq}_winter_storm (*DataArray*) – Number of days per period identified as winter storms. [days] description: {freq} number of days with snowfall accumulation above {thresh}.

Notes

Snowfall accumulation is estimated by the change in snow depth.

xclim.indicators.land._streamflow module

Streamflow indicator definitions.

`xclim.indicators.land._streamflow.base_flow_index(`*q: Union[DataArray, str] = 'q', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Base flow index. (realm: land)

Return the base flow index, defined as the minimum 7-day average flow divided by the mean flow.

This indicator will check for missing values according to the method "from_context". Based on indice *base_flow_index()*.

Parameters

- **q** (*str or DataArray*) – Rate of river discharge. Default : *ds.q*. [Required units : [discharge]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

base_flow_index (*DataArray*) – Base flow index description: Minimum 7-day average flow divided by the mean flow.

Notes

Let $\mathbf{q} = q_0, q_1, \ldots, q_n$ be the sequence of daily discharge and $\overline{\mathbf{q}}$ the mean flow over the period. The base flow index is given by:

$$\frac{\min(\mathrm{CMA}_7(\mathbf{q}))}{\overline{\mathbf{q}}}$$

where CMA_7 is the seven days moving average of the daily flow:

$$\mathrm{CMA}_7(q_i) = \frac{\sum_{j=i-3}^{i+3} q_j}{7}$$

`xclim.indicators.land._streamflow.doy_qmax(`*da: Union[DataArray, str] = 'da', *, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Day of year of the maximum. (realm: land)

This indicator will check for missing values according to the method "from_context". Based on indice *select_resample_op()*. With injected parameters: op=<function doymax at 0x7fea13c34f70>.

Parameters

- **da** (*str or DataArray*) – Input data. Default : *ds.da*.
- **freq** (*offset alias (string)*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **indexer** – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered. Default : None.

Returns

q{indexer}_doy_qmax (*DataArray*) – Day of the year of the maximum over {indexer} description: Day of the year of the maximum over {indexer}

`xclim.indicators.land._streamflow.doy_qmin`(*da: Union[DataArray, str] = 'da'*, ***, *freq: str = 'YS'*, *ds: Dataset = None*, ***indexer*) → DataArray

Day of year of the minimum. (realm: land)

This indicator will check for missing values according to the method "from_context". Based on indice *select_resample_op()*. With injected parameters: op=<function doymin at 0x7fea13c10040>.

Parameters

- **da** (*str or DataArray*) – Input data. Default : *ds.da*.
- **freq** (*offset alias (string)*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered. Default : None.

Returns

q{indexer}_doy_qmin (*DataArray*) – Day of the year of the minimum over {indexer} description: Day of the year of the minimum over {indexer}

`xclim.indicators.land._streamflow.fit`(*da: Union[DataArray, str] = 'da'*, ***, *dist: str = 'norm'*, *method: str = 'ML'*, *dim: str = 'time'*, *ds: Dataset = None*, ***fitkwargs*) → DataArray

Distribution parameters fitted over the time dimension. (realm: land)

Based on indice *fit()*.

Parameters

- **da** (*str or DataArray*) – Time series to be fitted along the time dimension. Default : *ds.da*.
- **dist** (*str*) – Name of the univariate distribution, such as beta, expon, genextreme, gamma, gumbel_r, lognorm, norm (see scipy.stats for full list). If the PWM method is used, only the following distributions are currently supported: 'expon', 'gamma', 'genextreme', 'genpareto', 'gumbel_r', 'pearson3', 'weibull_min'. Default : norm.
- **method** (*{'PWM', 'ML'}*) – Fitting method, either maximum likelihood (ML) or probability weighted moments (PWM), also called L-Moments. The PWM method is usually more robust to outliers. Default : ML.
- **dim** (*str*) – The dimension upon which to perform the indexing (default: "time"). Other arguments passed directly to `_fitstart()` and to the distribution's *fit*. Default : time.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

- **fitkwargs** – Default : None.

Returns
params (*DataArray*) – {dist} distribution parameters ({dist} parameters) cell_methods: time: fit description: Parameters of the {dist} distribution

Notes

Coordinates for which all values are NaNs will be dropped before fitting the distribution. If the array still contains NaNs, the distribution parameters will be returned as NaNs.

`xclim.indicators.land._streamflow.freq_analysis(`*da: Union[DataArray, str] = 'da', *, mode: str, t: int | Sequence[int], dist: str, window: int = 1, freq: str | None = None, ds: Dataset = None, **indexer*`)` → DataArray

Flow values for given return periods. (realm: land)

This indicator will check for missing values according to the method "skip". Based on indice *frequency_analysis()*.

Parameters

- **da** (*str or DataArray*) – Input data. Default : *ds.da*.
- **mode** (*{'min', 'max'}*) – Whether we are looking for a probability of exceedance (high) or a probability of non-exceedance (low). Default : *ds.da*.
- **t** (*number or sequence of numbers*) – Return period. The period depends on the resolution of the input data. If the input array's resolution is yearly, then the return period is in years. Default : *ds.da*.
- **dist** (*str*) – Name of the univariate distribution, such as *beta*, *expon*, *genextreme*, *gamma*, *gumbel_r*, *lognorm*, *norm* (see scipy.stats). Default : *ds.da*.
- **window** (*number*) – Averaging window length (days). Default : 1.
- **freq** (*offset alias (string)*) – Resampling frequency. If None, the frequency is assumed to be 'YS' unless the indexer is season='DJF', in which case *freq* would be set to *AS-DEC*. Default : None.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered. Default : None.

Returns
q{window}{mode (*r}{indexer} : DataArray*) – N-year return period {mode} {indexer} {window}-day flow [m^3 s-1] description: Streamflow frequency analysis for the {mode} {indexer} {window}-day flow estimated using the {dist} distribution.

`xclim.indicators.land._streamflow.rb_flashiness_index(`*q: Union[DataArray, str] = 'q', *, freq: str = 'YS', ds: Dataset = None*`)` → DataArray

Richards-Baker flashiness index. (realm: land)

Measures oscillations in flow relative to total flow, quantifying the frequency and rapidity of short term changes in flow, based on Baker et al. (2004; [baker2004]).

This indicator will check for missing values according to the method "from_context". Based on indice *rb_flashiness_index()*.

Parameters

- **q** (*str or DataArray*) – Rate of river discharge. Default : *ds.q*. [Required units : [discharge]]
- **freq** (*offset alias (string)*) – Resampling frequency. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

rbi (*DataArray*) – Richards-Baker flashiness index description: {freq} R-B Index, an index measuring the flashiness of flow.

Notes

Let $\mathbf{q} = q_0, q_1, \ldots, q_n$ be the sequence of daily discharge, the R-B Index is given by:

$$\frac{\sum_{i=1}^{n} |q_i - q_{i-1}|}{\sum_{i=1}^{n} q_i}$$

References

`xclim.indicators.land._streamflow.stats(`*da: Union[DataArray, str] = 'da', *, op: str, freq: str = 'YS', ds: Dataset = None, **indexer*`)` → DataArray

Statistic of the daily flow on a given period. (realm: land)

This indicator will check for missing values according to the method "any". Based on indice *select_resample_op()*.

Parameters

- **da** (*str or DataArray*) – Input data. Default : *ds.da*.
- **op** (*{'min', 'max', 'argmin', 'mean', 'var', 'argmax', 'sum', 'count', 'std'}*) – Reduce operation. Can either be a DataArray method or a function that can be applied to a DataArray. Default : *ds.da*.
- **freq** (*offset alias (string)*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling. Default : YS.
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.
- **indexer** – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered. Default : None.

Returns

q{indexer}{op (*r} : DataArray*) – {freq} {op} of {indexer} daily flow [m^3 s-1] description: {freq} {op} of {indexer} daily flow

xclim.indicators.seaIce package

Ice-related indicators

Submodules

xclim.indicators.seaIce._seaice module

Sea ice indicators

`xclim.indicators.seaIce._seaice.sea_ice_area`(*siconc: Union[DataArray, str] = 'siconc', areacello: Union[DataArray, str] = 'areacello', *, thresh: str = '15 pct', ds: Dataset = None*) → DataArray

Total sea ice area. (realm: seaIce)

Sea ice area measures the total sea ice covered area where sea ice concentration is above a threshold, usually set to 15%.

This indicator will check for missing values according to the method "skip". Based on indice `sea_ice_area()`.

Parameters

- **siconc** (*str or DataArray*) – Sea ice concentration (area fraction). Default : *ds.siconc*. [Required units : []]
- **areacello** (*str or DataArray*) – Grid cell area (usually over the ocean). Default : *ds.areacello*. [Required units : [area]]
- **thresh** (*quantity (string with units)*) – Minimum sea ice concentration for a grid cell to contribute to the sea ice extent. Default : 15 pct. [Required units : []]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

sea_ice_area (*DataArray*) – Sea ice area (sea_ice_area) [m2] cell_methods: lon: sum lat: sum description: The sum of ice-covered areas where sea ice concentration is at least {thresh}.

Notes

To compute sea ice area over a subregion, first mask or subset the input sea ice concentration data.

References

What is the difference between sea ice area and extent

`xclim.indicators.seaIce._seaice.sea_ice_extent`(*siconc: Union[DataArray, str] = 'siconc', areacello: Union[DataArray, str] = 'areacello', *, thresh: str = '15 pct', ds: Dataset = None*) → DataArray

Total sea ice extent. (realm: seaIce)

Sea ice extent measures the *ice-covered* area, where a region is considered ice-covered if its sea ice concentration is above a threshold usually set to 15%.

This indicator will check for missing values according to the method "skip". Based on indice *sea_ice_extent()*.

Parameters

- **siconc** (*str or DataArray*) – Sea ice concentration (area fraction). Default : *ds.siconc*. [Required units : []]
- **areacello** (*str or DataArray*) – Grid cell area. Default : *ds.areacello*. [Required units : [area]]
- **thresh** (*quantity (string with units)*) – Minimum sea ice concentration for a grid cell to contribute to the sea ice extent. Default : 15 pct. [Required units : []]
- **ds** (*Dataset, optional*) – A dataset with the variables given by name. Default : None.

Returns

sea_ice_extent (*DataArray*) – Sea ice extent (sea_ice_extent) [m2] cell_methods: lon: sum lat: sum description: The sum of ocean areas where sea ice concentration is at least {thresh}.

Notes

To compute sea ice area over a subregion, first mask or subset the input sea ice concentration data.

References

What is the difference between sea ice area and extent

xclim.indices package

Indices library

This module contains climate indices functions operating on *xarray.DataArray*. Most of these functions operate on daily time series, but might accept other sampling frequencies as well. All functions perform units checks to make sure that inputs have the expected dimensions (for example have units of temperature, whether it is celsius, kelvin or fahrenheit), and set the *units* attribute of the output DataArray.

The *calendar*, *fwi*, *generic*, *helpers*, *run_length* and *stats* submodules provide helpers to simplify the implementation of the indices.

Note: Indices functions do not perform missing value checks, and usually do not set CF-Convention attributes (long_name, standard_name, description, cell_methods, etc.). These functionalities are provided by `xclim.indicators.Indicator` instances found in the *xclim.indicators.atmos*, *xclim.indicators.land* and *xclim.indicators.seaIce* modules, documented in *Climate indicators*.

Submodules

xclim.indices._agro module

`xclim.indices._agro.biologically_effective_degree_days(`*tasmin: xarray.DataArray, tasmax: xarray.DataArray, lat: xarray.DataArray | None = None, thresh_tasmin: str = '10 degC', method: str = 'gladstones', low_dtr: str = '10 degC', high_dtr: str = '13 degC', max_daily_degree_days: str = '9 degC', start_date: DayOfYearStr = '04-01', end_date: DayOfYearStr = '11-01', freq: str = 'YS'*`)` → xarray.DataArray

Biologically effective growing degree days.

Growing-degree days with a base of 10°C and an upper limit of 19°C and adjusted for latitudes between 40°N and 50°N for April to October (Northern Hemisphere; October to April in Southern Hemisphere). A temperature range adjustment also promotes small and large swings in daily temperature range. Used as a heat-summation metric in viticulture agroclimatology.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **lat** (*xarray.DataArray, optional*) – Latitude coordinate.
- **thresh_tasmin** (*str*) – The minimum temperature threshold.
- **method** (*{"gladstones", "icclim", "jones"}*) – The formula to use for the calculation. The "gladstones" integrates a daily temperature range and latitude coefficient. End_date should be "11-01". The "icclim" method ignores daily temperature range and latitude coefficient. End date should be "10-01". The "jones" method integrates axial tilt, latitude, and day-of-year on coefficient. End_date should be "11-01".
- **low_dtr** (*str*) – The lower bound for daily temperature range adjustment (default: 10°C).
- **high_dtr** (*str*) – The higher bound for daily temperature range adjustment (default: 13°C).
- **max_daily_degree_days** (*str*) – The maximum amount of biologically effective degrees days that can be summed daily.
- **start_date** (*DayOfYearStr*) – The hemisphere-based start date to consider (north = April, south = October).
- **end_date** (*DayOfYearStr*) – The hemisphere-based start date to consider (north = October, south = April). This date is non-inclusive.
- **freq** (*str*) – Resampling frequency (default: "YS"; For Southern Hemisphere, should be "AS-JUL").

Returns

xarray.DataArray – Biologically effective growing degree days (BEDD).

Warning: Lat coordinate must be provided if method is "gladstones" or "jones".

Notes

The tasmax ceiling of 19°C is assumed to be the max temperature beyond which no further gains from daily temperature occur. Indice originally published in [Gladstones1992].

Let TX_i and TN_i be the daily maximum and minimum temperature at day i, lat the latitude of the point of interest, $degdays_{max}$ the maximum amount of degrees that can be summed per day (typically, 9). Then the sum of daily biologically effective growing degree day (BEDD) units between 1 April and 31 October is:

$$BEDD_i = \sum_{i=\text{April 1}}^{\text{October 31}} min\left(\left(max\left(\frac{TX_i + TN_i)}{2} - 10, 0\right) * k\right) + TR_{adj}, degdays_{max}\right)$$

$$TR_{adj} = f(TX_i, TN_i) = \begin{cases} 0.25(TX_i - TN_i - 13), & \text{if } (TX_i - TN_i) > 13 \\ 0, & \text{if } 10 < (TX_i - TN_i) < 13 \\ 0.25(TX_i - TN_i - 10), & \text{if } (TX_i - TN_i) < 10 \end{cases}$$

$$k = f(lat) = 1 + \left(\frac{|lat|}{50} * 0.06, \text{if } 40 < |lat| < 50, \text{else } 0\right)$$

A second version of the BEDD (*method="icclim"*) does not consider TR_{adj} and k and employs a different end date (30 September) ([ECAD]). The simplified formula is as follows:

$$BEDD_i = \sum_{i=\text{April 1}}^{\text{September 30}} min\left(max\left(\frac{TX_i + TN_i)}{2} - 10, 0\right), degdays_{max}\right)$$

References

`xclim.indices._agro.cool_night_index`(*tasmin: DataArray*, *lat: DataArray*, *freq: str = 'YS'*) → DataArray

Cool Night Index.

Mean minimum temperature for September (northern hemisphere) or March (Southern hemishere). Used in calculating the Géoviticulture Multicriteria Classification System ([Tonietto&Carbonneau2004]_).

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **lat** (*xarray.DataArray, optional*) – Latitude coordinate.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [degC] – Mean of daily minimum temperature for month of interest.

Notes

Given that this indice only examines September and March months, it is possible to send in DataArrays containing only these timesteps. Users should be aware that due to the missing values checks in wrapped Indicators, datasets that are missing several months will be flagged as invalid. This check can be ignored by setting the following context:

Examples

```
>>> with xclim.set_options(
...     check_missing="skip", data_validation="log"
... ):
...     cni = xclim.atmos.cool_night_index(...)
...
```

References

`xclim.indices._agro.corn_heat_units(`*tasmin: DataArray*, *tasmax: DataArray*, *thresh_tasmin: str = '4.44 degC'*, *thresh_tasmax: str = '10 degC'*`)` → DataArray

Corn heat units.

Temperature-based index used to estimate the development of corn crops. Formula adapted from [BootsmaTremblay&Filion1999]_.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – The minimum temperature threshold needed for corn growth.
- **thresh_tasmax** (*str*) – The maximum temperature threshold needed for corn growth.

Returns

xarray.DataArray, [dimensionless] – Daily corn heat units.

Notes

Formula used in calculating the Corn Heat Units for the Agroclimatic Atlas of Quebec [Audet&al2012]_.

The thresholds of 4.44°C for minimum temperatures and 10°C for maximum temperatures were selected following the assumption that no growth occurs below these values.

Let TX_i and TN_i be the daily maximum and minimum temperature at day i. Then the daily corn heat unit is:

$$CHU_i = \frac{YX_i + YN_i}{2}$$

with

$$\begin{aligned} YX_i &= 3.33(TX_i - 10) - 0.084(TX_i - 10)^2, & \text{if } TX_i > 10C \\ YN_i &= 1.8(TN_i - 4.44), & \text{if } TN_i > 4.44C \end{aligned}$$

where YX_i and YN_i is 0 when $TX_i \leq 10C$ and $TN_i \leq 4.44C$, respectively.

References

`xclim.indices._agro.dry_spell_frequency`(*pr: DataArray, thresh: str = '1.0 mm', window: int = 3, freq: str = 'YS', op: str = 'sum'*) → DataArray

Return the number of dry periods of n days and more.

Periods during which the accumulated or maximal daily precipitation amount on a window of n days is under threshold.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **thresh** (*str*) – Precipitation amount under which a period is considered dry. The value against which the threshold is compared depends on *op* .
- **window** (*int*) – Minimum length of the spells.
- **freq** (*str*) – Resampling frequency.
- **op** (*{"sum","max"}*) – Operation to perform on the window. Default is "sum", which checks that the sum of accumulated precipitation over the whole window is less than the threshold. "max" checks that the maximal daily precipitation amount within the window is less than the threshold. This is the same as verifying that each individual day is below the threshold.

Returns

xarray.DataArray – The {freq} number of dry periods of minimum {window} days.

Examples

```
>>> pr = xr.open_dataset(path_to_pr_file).pr
>>> dry_spell_frequency(pr=pr, op="sum")
>>> dry_spell_frequency(pr=pr, op="max")
```

`xclim.indices._agro.dry_spell_total_length`(*pr: DataArray, thresh: str = '1.0 mm', window: int = 3, op: str = 'sum', freq: str = 'YS', **indexer*) → DataArray

Total length of dry spells.

Total number of days in dry periods of a minimum length, during which the maximum or accumulated precipitation within a window of the same length is under a threshold.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **thresh** (*str*) – Accumulated precipitation value under which a period is considered dry.
- **window** (*int*) – Number of days when the maximum or accumulated precipitation is under threshold.
- **op** (*{"max", "sum"}*) – Reduce operation.
- **freq** (*str*) – Resampling frequency.

- **indexer** – Indexing parameters to compute the indicator on a temporal subset of the data. It accepts the same arguments as `xclim.indices.generic.select_time()`. Indexing is done after finding the dry days, but before finding the spells.

Returns
xarray.DataArray – The {freq} total number of days in dry periods of minimum {window} days.

Notes

The algorithm assumes days before and after the timeseries are "wet", meaning that the condition for being considered part of a dry spell is stricter on the edges. For example, with *window=3* and *op='sum'*, the first day of the series is considered part of a dry spell only if the accumulated precipitation within the first 3 days is under the threshold. In comparison, a day in the middle of the series is considered part of a dry spell if any of the three 3-day periods of which it is part are considered dry (so a total of five days are included in the computation, compared to only 3.)

`xclim.indices._agro.effective_growing_degree_days(`*tasmax: DataArray, tasmin: DataArray, *, thresh: str = '5 degC', method: str = 'bootsma', after_date: DayOfYearStr = '07-01', dim: str = 'time', freq: str = 'YS'*) → DataArray

Effective growing degree days.

Growing degree days based on a dynamic start and end of the growing season, as defined in [BootsmaGameda&McKenney2005]_.

Parameters

- **tasmax** (*xr.DataArray*) – Daily mean temperature.
- **tasmin** (*xr.DataArray*) – Daily minimum temperature.
- **thresh** (*str*) – The minimum temperature threshold.
- **method** (*{"bootsma", "qian"}*) – The window method used to determine the temperature-based start date. For "bootsma", the start date is defined as 10 days after the average temperature exceeds a threshold (5 degC). For "qian", the start date is based on a weighted 5-day rolling average, based on *qian_weighted_mean_average()*.
- **after_date** (*str*) – Date of the year after which to look for the first frost event. Should have the format '%m-%d'.
- **dim** (*str*) – Time dimension.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

Notes

The effective growing degree days for a given year $EGDD_i$ can be calculated as follows:

The end date is determined as the day preceding the first day with minimum temperature below 0 degC.

References

`xclim.indices._agro.huglin_index(`*tas: DataArray, tasmax: DataArray, lat: DataArray, thresh: str = '10 degC', method: str = 'smoothed', start_date: DayOfYearStr = '04-01', end_date: DayOfYearStr = '10-01', freq: str = 'YS'*`)` → DataArray

Huglin Heliothermal Index.

Growing-degree days with a base of 10°C and adjusted for latitudes between 40°N and 50°N for April to September (Northern Hemisphere; October to March in Southern Hemisphere). Originally proposed in [Huglin1978]. Used as a heat-summation metric in viticulture agroclimatology.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **lat** (*xarray.DataArray*) – Latitude coordinate.
- **thresh** (*str*) – The temperature threshold.
- **method** (*{"smoothed", "icclim", "jones"}*) – The formula to use for the latitude coefficient calculation.
- **start_date** (*DayOfYearStr*) – The hemisphere-based start date to consider (north = April, south = October).
- **end_date** (*DayOfYearStr*) – The hemisphere-based start date to consider (north = October, south = April). This date is non-inclusive.
- **freq** (*str*) – Resampling frequency (default: "YS"; For Southern Hemisphere, should be "AS-JUL").

Returns

xarray.DataArray, [unitless] – Huglin heliothermal index (HI).

Notes

Let TX_i and TG_i be the daily maximum and mean temperature at day i and T_{thresh} the base threshold needed for heat summation (typically, 10 degC). A day-length multiplication, k, based on latitude, lat, is also considered. Then the Huglin heliothermal index for dates between 1 April and 30 September is:

$$HI = \sum_{i=\text{April 1}}^{\text{September 30}} \left(\frac{TX_i + TG_i)}{2} - T_{thresh} \right) * k$$

For the *smoothed* method, the day-length multiplication factor, k, is calculated as follows:

$$k = f(lat) = \begin{cases} 1, & \text{if } |lat| <= 40 \\ 1 + ((abs(lat) - 40)/10) * 0.06, & \text{if } 40 < |lat| <= 50 \\ NaN, & \text{if } |lat| > 50 \end{cases}$$

For compatibility with ICCLIM, *end_ date* should be set to *11-01*, *method* should be set to *icclim*. The day-length multiplication factor, k, is calculated as follows:

$$k = f(lat) = \begin{cases} 1.0, & \text{if } |lat| <= 40 \\ 1.02, & \text{if } 40 < |lat| <= 42 \\ 1.03, & \text{if } 42 < |lat| <= 44 \\ 1.04, & \text{if } 44 < |lat| <= 46 \\ 1.05, & \text{if } 46 < |lat| <= 48 \\ 1.06, & \text{if } 48 < |lat| <= 50 \\ NaN, & \text{if } |lat| > 50 \end{cases}$$

A more robust day-length calculation based on latitude, calendar, day-of-year, and obliquity is available with *method="jones"*. See: `xclim.indices.generic.day_lengths()` or [Hall&Jones2010]_ for more information.

References

`xclim.indices._agro.latitude_temperature_index`(*tas: DataArray*, *lat: DataArray*, *lat_factor: float = 75*, *freq: str = 'YS'*) → DataArray

Latitude-Temperature Index.

Mean temperature of the warmest month with a latitude-based scaling factor ([Jackson&Cherry1988]_). Used for categorizing wine-growing regions.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **lat** (*xarray.DataArray*) – Latitude coordinate.
- **lat_factor** (*float*) – Latitude factor. Maximum poleward latitude. Default: 75.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [unitless] – Latitude Temperature Index.

Notes

The latitude factor of *75* is provided for examining the poleward expansion of wine-growing climates under scenarios of climate change (modified from [Kenny&Shao1992]_). For comparing 20th century/observed historical records, the original scale factor of *60* is more appropriate.

Let Tn_j be the average temperature for a given month j, lat_f be the latitude factor, and lat be the latitude of the area of interest. Then the Latitude-Temperature Index (LTI) is:

$$LTI = max(TN_j : j = 1..12)(lat_f - |lat|)$$

References

`xclim.indices._agro.qian_weighted_mean_average(`*tas: DataArray*, *dim: str = 'time'*`)` → DataArray

Binomial smoothed, five-day weighted mean average temperature.

Calculates a five-day weighted moving average with emphasis on temperatures closer to day of interest.

Parameters

- **tas** (*xr.DataArray*) – Daily mean temperature.
- **dim** (*str*) – Time dimension.

Returns

xr.DataArray – Binomial smoothed, five-day weighted mean average temperature.

Notes

Qian Modified Weighted Mean Indice originally proposed in [Qian&al2009]_, based on [BootsmaGameda&McKenney2005]_.

Let X_n be the average temperature for day n and X_t be the daily mean temperature on day t. Then the weighted mean average can be calculated as follows:

$$\overline{X}_n = \frac{X_{n-2} + 4X_{n-1} + 6X_n + 4X_{n+1} + X_{n+2}}{16}$$

References

`xclim.indices._agro.water_budget(`*pr: xarray.DataArray*, *evspsblpot: xarray.DataArray | None = None*, *tasmin: xarray.DataArray | None = None*, *tasmax: xarray.DataArray | None = None*, *tas: xarray.DataArray | None = None*, *lat: xarray.DataArray | None = None*, *method: str = 'BR65'*`)` → xarray.DataArray

Precipitation minus potential evapotranspiration.

Precipitation minus potential evapotranspiration as a measure of an approximated surface water budget, where the potential evapotranspiration can be calculated with a given method.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **evspsblpot** (*xarray.DataArray*) – Potential evapotranspiration
- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **lat** (*xarray.DataArray*) – Latitude, needed if evspsblpot is not given.
- **method** (*str*) – Method to use to calculate the potential evapotranspiration.

Notes

Available methods are listed in the description of xclim.indicators.atmos.potential_evapotranspiration.

Returns
xarray.DataArray, – Precipitation minus potential evapotranspiration.

xclim.indices._anuclim module

`xclim.indices._anuclim.isothermality(`*tasmin: DataArray*, *tasmax: DataArray*, *freq: str = 'YS'*`)` → DataArray

Isothermality.

The mean diurnal range divided by the annual temperature range.

Parameters

- **tasmin** (*xarray.DataArray*) – Average daily minimum temperature at daily, weekly, or monthly frequency.
- **tasmax** (*xarray.DataArray*) – Average daily maximum temperature at daily, weekly, or monthly frequency.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [%] – Isothermality

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the output with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices._anuclim.prcptot(`*pr: DataArray*, *thresh: str = '0 mm/d'*, *freq: str = 'YS'*`)` → DataArray

Accumulated total precipitation.

Parameters

- **pr** (*xarray.DataArray*) – Total precipitation flux [mm d-1], [mm week-1], [mm month-1] or similar.
- **thresh** (*str*) – Threshold over which precipitation starts being cumulated.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [length] – Total {freq} precipitation.

`xclim.indices._anuclim.prcptot_warmcold_quarter(`*pr: DataArray*, *tas: DataArray*, *op: Optional[str] = None*, *freq: str = 'YS'*`)` → DataArray

ANUCLIM Total precipitation of warmest/coldest quarter.

The warmest (or coldest) quarter of the year is determined, and the total precipitation of this period is calculated. If the input data frequency is daily (“D) or weekly (“W”), quarters are defined as 13-week periods, otherwise are 3 months.

Parameters

- **pr** (*xarray.DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency.
- **tas** (*xarray.DataArray*) – Mean temperature at daily, weekly, or monthly frequency.
- **op** (*{'warmest', 'coldest'}*) – Operation to perform: 'warmest' calculate for the warmest quarter ; 'coldest' calculate for the coldest quarter.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray (*[mm]*) – Precipitation of {op} quarter

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices._anuclim.prcptot_wetdry_period(`*pr: DataArray*, ***, *op: str*, *freq: str = 'YS'*`)` → DataArray

ANUCLIM precipitation of the wettest/driest day, week, or month, depending on the time step.

Parameters

- **pr** (*xarray.DataArray*) – Total precipitation flux [mm d-1], [mm week-1], [mm month-1] or similar.
- **op** (*{'wettest', 'driest'}*) – Operation to perform : 'wettest' calculate the wettest period ; 'driest' calculate the driest period.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [length] – Precipitation of {op} period

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices._anuclim.prcptot_wetdry_quarter(`*pr: DataArray*, *op: Optional[str] = None*, *freq: str = 'YS'*`)` → DataArray

ANUCLIM Total precipitation of wettest/driest quarter.

The wettest (or driest) quarter of the year is determined, and the total precipitation of this period is calculated. If the input data frequency is daily ("D") or weekly ("W") quarters are defined as 13-week periods, otherwise are 3 months.

Parameters

- **pr** (*xarray.DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency.

- **op** (*{'wettest', 'driest'}*) – Operation to perform : 'wettest' calculate the wettest quarter ; 'driest' calculate the driest quarter.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [length] – Precipitation of {op} quarter

Examples

The following would compute for each grid cell of file *pr.day.nc* the annual wettest quarter total precipitation:

```
>>> from xclim.indices import prcptot_wetdry_quarter
>>> p = xr.open_dataset(path_to_pr_file)
>>> pr_warm_qrt = prcptot_wetdry_quarter(pr=p.pr, op="wettest")
```

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices._anuclim.precip_seasonality`(*pr: DataArray, freq: str = 'YS'*) → DataArray

ANUCLIM Precipitation Seasonality (C of V).

The annual precipitation Coefficient of Variation (C of V) expressed in percent. Calculated as the standard deviation of precipitation values for a given year expressed as a percentage of the mean of those values.

Parameters

- **pr** (*xarray.DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency. Units need to be defined as a rate (e.g. mm d-1, mm week-1).
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [%] – Precipitation coefficient of variation

Examples

The following would compute for each grid cell of file *pr.day.nc* the annual precipitation seasonality:

```
>>> import xclim.indices as xci
>>> p = xr.open_dataset(path_to_pr_file).pr
>>> pday_seasonality = xci.precip_seasonality(p)
>>> p_weekly = xci.precip_accumulation(p, freq="7D")
```

# Input units need to be a rate >>> p_weekly.attrs["units"] = "mm/week" >>> pweek_seasonality = xci.precip_seasonality(p_weekly)

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

If input units are in mm s-1 (or equivalent) values are converted to mm/day to avoid potentially small denominator values.

`xclim.indices._anuclim.temperature_seasonality`(*tas: DataArray*, *freq: str = 'YS'*) → DataArray

ANUCLIM temperature seasonality (coefficient of variation).

The annual temperature coefficient of variation expressed in percent. Calculated as the standard deviation of temperature values for a given year expressed as a percentage of the mean of those temperatures.

Parameters

- **tas** (*xarray.DataArray*) – Mean temperature at daily, weekly, or monthly frequency.
- **freq** (*str*) – Resampling frequency.

Returns

- *xarray.DataArray, [%]* – Mean temperature coefficient of variation
- **freq** (*str*) – Resampling frequency.

Examples

The following would compute for each grid cell of file *tas.day.nc* the annual temperature seasonality:

```
>>> import xclim.indices as xci
>>> t = xr.open_dataset(path_to_tas_file).tas
>>> tday_seasonality = xci.temperature_seasonality(t)
>>> t_weekly = xci.tg_mean(t, freq="7D")
>>> tweek_seasonality = xci.temperature_seasonality(t_weekly)
```

Notes

For this calculation, the mean in degrees Kelvin is used. This avoids the possibility of having to divide by zero, but it does mean that the values are usually quite small.

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices._anuclim.tg_mean_warmcold_quarter`(*tas: DataArray*, *op: Optional[str] = None*, *freq: str = 'YS'*) → DataArray

ANUCLIM Mean temperature of warmest/coldest quarter.

The warmest (or coldest) quarter of the year is determined, and the mean temperature of this period is calculated. If the input data frequency is daily ("D") or weekly ("W"), quarters are defined as 13-week periods, otherwise as 3 months.

Parameters

- **tas** (*xarray.DataArray*) – Mean temperature at daily, weekly, or monthly frequency.
- **op** (*str {'warmest', 'coldest'}*) – Operation to perform: 'warmest' calculate the warmest quarter; 'coldest' calculate the coldest quarter.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same as tas] – Mean temperature of {op} quarter

Examples

The following would compute for each grid cell of file *tas.day.nc* the annual temperature warmest quarter mean temperature:

```
>>> import xclim.indices as xci
>>> t = xr.open_dataset(path_to_tas_file)
>>> t_warm_qrt = xci.tg_mean_warmcold_quarter(tas=t.tas, op="warmest")
```

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

`xclim.indices._anuclim.tg_mean_wetdry_quarter`(*tas: DataArray*, *pr: DataArray*, *op: Optional[str] = None*, *freq: str = 'YS'*) → DataArray

ANUCLIM Mean temperature of wettest/driest quarter.

The wettest (or driest) quarter of the year is determined, and the mean temperature of this period is calculated. If the input data frequency is daily ("D") or weekly ("W"), quarters are defined as 13-week periods, otherwise are 3 months.

Parameters

- **tas** (*xarray.DataArray*) – Mean temperature at daily, weekly, or monthly frequency.
- **pr** (*xarray.DataArray*) – Total precipitation rate at daily, weekly, or monthly frequency.
- **op** (*{'wettest', 'driest'}*) – Operation to perform: 'wettest' calculate for the wettest quarter; 'driest' calculate for the driest quarter.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same as tas] – Mean temperature of {op} quarter

Notes

According to the ANUCLIM user-guide https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6), input values should be at a weekly (or monthly) frequency. However, the xclim.indices implementation here will calculate the result with input data with daily frequency as well. As such weekly or monthly input values, if desired, should be calculated prior to calling the function.

xclim.indices._conversion module

`xclim.indices._conversion.clausius_clapeyron_scaled_precipitation(`*delta_tas: DataArray, pr_baseline: DataArray, cc_scale_factor: float = 1.07*`)` → DataArray

Scale precipitation according to the Clausius-Clapeyron relation.

Parameters

- **delta_tas** (*xarray.DataArray*) – Difference in temperature between a baseline climatology and another climatology.
- **pr_baseline** (*xarray.DataArray*) – Baseline precipitation to adjust with Clausius-Clapeyron.
- **cc_scale_factor** (*float (default = 1.07)*) – Clausius Clapeyron scale factor.

Returns

DataArray – Baseline precipitation scaled to other climatology using Clausius-Clapeyron relationship.

Notes

The Clausius-Clapeyron equation for water vapor under typical atmospheric conditions states that the saturation water vapor pressure e_s changes approximately exponentially with temperature

$$\frac{\mathrm{d}e_s(T)}{\mathrm{d}T} \approx 1.07 e_s(T)$$

This function assumes that precipitation can be scaled by the same factor.

Warning: Make sure that *delta_tas* is computed over a baseline compatible with *pr_baseline*. So for example, if *delta_tas* is the climatological difference between a baseline and a future period, then *pr_baseline* should be precipitations over a period within the same baseline.

`xclim.indices._conversion.heat_index(`*tasmax: DataArray, hurs: DataArray*`)` → DataArray

Daily heat index.

Perceived temperature after relative humidity is taken into account ([Blazejczyk2012]). The index is only valid for temperatures above 20°C.

Parameters

- **tasmax** (*xr.DataArray*) – Maximum daily temperature.
- **hurs** (*xr.DataArray*) – Relative humidity.

Returns

xr.DataArray, [time][temperature] – Heat index for days with temperature above 20°C.

References

Notes

While both the humidex and the heat index are calculated using dew point, the humidex uses a dew point of 7 °C (45 °F) as a base, whereas the heat index uses a dew point base of 14 °C (57 °F). Further, the heat index uses heat balance equations which account for many variables other than vapor pressure, which is used exclusively in the humidex calculation.

`xclim.indices._conversion.humidex(`*tas: xr.DataArray*, *tdps: xr.DataArray | None = None*, *hurs: xr.DataArray | None = None*`)` → xr.DataArray

Humidex index.

The humidex indicates how hot the air feels to an average person, accounting for the effect of humidity. It can be loosely interpreted as the equivalent perceived temperature when the air is dry.

Parameters

- **tas** (*xarray.DataArray*) – Air temperature.
- **tdps** (*xarray.DataArray,*) – Dewpoint temperature.
- **hurs** (*xarray.DataArray*) – Relative humidity.

Returns

xarray.DataArray, [temperature] – The humidex index.

Notes

The humidex is usually computed using hourly observations of dry bulb and dewpoint temperatures. It is computed using the formula based on [masterton79]:

$$T + \frac{5}{9}\left[e - 10\right]$$

where T is the dry bulb air temperature (°C). The term e can be computed from the dewpoint temperature $T_{dewpoint}$ in °K:

$$e = 6.112 \times \exp(5417.7530\left({\frac{1}{273.16}}-{\frac{1}{T_{\text{dewpoint}}}}\right)$$

where the constant 5417.753 reflects the molecular weight of water, latent heat of vaporization, and the universal gas constant ([mekis15]). Alternatively, the term e can also be computed from the relative humidity h expressed in percent using [sirangelo20]:

$$e = \frac{h}{100} \times 6.112 * 10^{7.5 T/(T + 237.7)}.$$

The humidex *comfort scale* ([eccc]) can be interpreted as follows:

- 20 to 29 : no discomfort;
- 30 to 39 : some discomfort;
- 40 to 45 : great discomfort, avoid exertion;
- 46 and over : dangerous, possible heat stroke;

Please note that while both the humidex and the heat index are calculated using dew point, the humidex uses a dew point of 7 °C (45 °F) as a base, whereas the heat index uses a dew point base of 14 °C (57 °F). Further, the heat index uses heat balance equations which account for many variables other than vapor pressure, which is used exclusively in the humidex calculation.

References

`xclim.indices._conversion.mean_radiant_temperature(`*rsds: DataArray*, *rsus: DataArray*, *rlds: DataArray*, *rlus: DataArray*, *stat: str = 'average'*`)` → DataArray

Mean radiant temperature.

The mean radiant temperature is the incidence of radiation on the body from all directions. WARNING: There are some issues in the calculation of mrt in polar regions.

Parameters

- **rsds** (*xr.DataArray*) – Surface Downwelling Shortwave Radiation
- **rsus** (*xr.DataArray*) – Surface Upwelling Shortwave Radiation
- **rlds** (*xr.DataArray*) – Surface Downwelling Longwave Radiation
- **rlus** (*xr.DataArray*) – Surface Upwelling Longwave Radiation
- **stat** (*{'average', 'instant', 'sunlit'}*) – Which statistic to apply. If "average", the average of the cosine of the solar zenith angle is calculated. If "instant", the instantaneous cosine of the solar zenith angle is calculated. If "sunlit", the cosine of the solar zenith angle is calculated during the sunlit period of each interval. If "instant", the instantaneous cosine of the solar zenith angle is calculated. This is necessary if mrt is not None.

Returns

xarray.DataArray, [K] – Mean radiant temperature

Notes

This code was inspired by the *thermofeel* package.

References

Di Napoli, C., Hogan, R.J. & Pappenberger, F. Mean radiant temperature from global-scale numerical weather prediction models. Int J Biometeorol 64, 1233–1245 (2020). https://doi.org/10.1007/s00484-020-01900-5 Brimicombe , C., Di Napoli, C., Quintino, T., Pappenberger, F., Cornforth, R. and Cloke, H., 2021 thermofeel: a python thermal comfort indices library, https://doi.org/10.21957/mp6v-fd16

`xclim.indices._conversion.potential_evapotranspiration(`*tasmin: xr.DataArray | None = None*, *tasmax: xr.DataArray | None = None*, *tas: xr.DataArray | None = None*, *lat: xr.DataArray | None = None*, *method: str = 'BR65'*, *peta: float | None = 0.00516409319477*, *petb: float | None = 0.0874972822289*`)` → xr.DataArray

Potential evapotranspiration.

The potential for water evaporation from soil and transpiration by plants if the water supply is sufficient, according to a given method.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **lat** (*xarray.DataArray, optional*) – Latitude. If not given, it is sought on tasmin or tas with cf-xarray.
- **method** (*{"baierrobertson65", "BR65", "hargreaves85", "HG85", "thornthwaite48", "TW48", "mcguinnessbordne05", "MB05"}*) – Which method to use, see notes.
- **peta** (*float*) – Used only with method MB05 as a for calculation of PET, see Notes section. Default value resulted from calibration of PET over the UK.
- **petb** (*float*) – Used only with method MB05 as b for calculation of PET, see Notes section. Default value resulted from calibration of PET over the UK.

Returns

xarray.DataArray

Notes

Available methods are:

- "baierrobertson65" or "BR65", based on [BaierRobertson1965]. Requires tasmin and tasmax, daily [D] freq.
- "hargreaves85" or "HG85", based on [Hargreaves1985]. Requires tasmin and tasmax, daily [D] freq. (optional: tas can be given in addition of tasmin and tasmax).
- "mcguinnessbordne05" or "MB05", based on [Tanguy2018]. Requires tas, daily [D] freq, with latitudes 'lat'.
- "thornthwaite48" or "TW48", based on [Thornthwaite1948]. Requires tasmin and tasmax, monthly [MS] or daily [D] freq. (optional: tas can be given instead of tasmin and tasmax).

The McGuinness-Bordne [McGuinness1972] equation is:

$$PET[mmday^{-1}] = a * \frac{S_0}{\lambda}T_a + b * S_0\lambda$$

where a and b are empirical parameters; S_0 is the extraterrestrial radiation [MJ m-2 day-1], assuming a solar constant of 1367 W m-2;
lambda is the latent heat of vaporisation [MJ kg-1] and T_a is the air temperature [°C]. The equation was originally derived for the USA, with $a = 0.0147$ and $b = 0.07353$. The default parameters used here are calibrated for the UK, using the method described in [Tanguy2018].

Methods "BR65", "HG85" and "MB05" use an approximation of the extraterrestrial radiation. See `extraterrestrial_solar_radiation()`.

References

`xclim.indices._conversion.rain_approximation`(*pr: DataArray, tas: DataArray, thresh: str = '0 degC', method: str = 'binary'*) → DataArray

Rainfall approximation from total precipitation and temperature.

Liquid precipitation estimated from precipitation and temperature according to a given method. This is a convenience method based on *snowfall_approximation()*, see the latter for details.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.

- **tas** (*xarray.DataArray, optional*) – Mean, maximum, or minimum daily temperature.
- **thresh** (*str,*) – Threshold temperature, used by method "binary".
- **method** (*{"binary", "brown", "auer"}*) – Which method to use when approximating snowfall from total precipitation. See notes.

Returns
xarray.DataArray, [same units as pr] – Liquid precipitation rate.

Notes

This method computes the snowfall approximation and subtracts it from the total precipitation to estimate the liquid rain precipitation.

See also:

`xclim.indices.snowfall_approximation()`

`xclim.indices._conversion.relative_humidity(`*tas: DataArray, tdps: Optional[DataArray] = None, huss: Optional[DataArray] = None, ps: Optional[DataArray] = None, ice_thresh: Optional[str] = None, method: str = 'sonntag90', invalid_values: str = 'clip'*`)` → DataArray

Relative humidity.

Compute relative humidity from temperature and either dewpoint temperature or specific humidity and pressure through the saturation vapor pressure.

Parameters

- **tas** (*xr.DataArray*) – Temperature array
- **tdps** (*xr.DataArray*) – Dewpoint temperature, if specified, overrides huss and ps.
- **huss** (*xr.DataArray*) – Specific humidity.
- **ps** (*xr.DataArray*) – Air Pressure.
- **ice_thresh** (*str*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water. Does nothing if 'method' is "bohren98".
- **method** (*{"bohren98", "goffgratch46", "sonntag90", "tetens30", "wmo08"}*) – Which method to use, see notes of this function and of *saturation_vapor_pressure*.
- **invalid_values** (*{"clip", "mask", None}*) – What to do with values outside the 0-100 range. If "clip" (default), clips everything to 0 - 100, if "mask", replaces values outside the range by np.nan, and if *None*, does nothing.

Returns
xr.DataArray, [%] – Relative humidity.

Notes

In the following, let T, T_d, q and p be the temperature, the dew point temperature, the specific humidity and the air pressure.

For the "bohren98" method : This method does not use the saturation vapor pressure directly, but rather uses an approximation of the ratio of $\frac{e_{sat}(T_d)}{e_{sat}(T)}$. With L the enthalpy of vaporization of water and R_w the gas constant for water vapor, the relative humidity is computed as:

$$RH = e^{\frac{-L(T-T_d)}{R_w T T_d}}$$

From [BohrenAlbrecht1998], formula taken from [Lawrence2005]. $L = 2.5 \times 10^{-6}$ J kg-1, exact for $T = 273.15$ K, is used.

Other methods: With w, w_{sat}, e_{sat} the mixing ratio, the saturation mixing ratio and the saturation vapor pressure. If the dewpoint temperature is given, relative humidity is computed as:

$$RH = 100\frac{e_{sat}(T_d)}{e_{sat}(T)}$$

Otherwise, the specific humidity and the air pressure must be given so relative humidity can be computed as:

$$RH = 100\frac{w}{w_{sat}} w = \frac{q}{1-q} w_{sat} = 0.622\frac{e_{sat}}{P - e_{sat}}$$

The methods differ by how e_{sat} is computed. See the doc of `xclim.core.utils.saturation_vapor_pressure()`.

References

`xclim.indices._conversion.saturation_vapor_pressure(`*tas: DataArray*, *ice_thresh: Optional[str] = None*, *method: str = 'sonntag90'*`)` → DataArray

Saturation vapor pressure from temperature.

Parameters

- **tas** (*xr.DataArray*) – Temperature array.
- **ice_thresh** (*str*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water.
- **method** (*{"goffgratch46", "sonntag90", "tetens30", "wmo08", "its90"}*) – Which method to use, see notes.

Returns

xarray.DataArray, [Pa] – Saturation vapor pressure.

Notes

In all cases implemented here $log(e_{sat})$ is an empirically fitted function (usually a polynomial) where coefficients can be different when ice is taken as reference instead of water. Available methods are:

- "goffgratch46" or "GG46", based on [goffgratch46], values and equation taken from [voemel].
- "sonntag90" or "SO90", taken from [sonntag90].
- "tetens30" or "TE30", based on [tetens30], values and equation taken from [voemel].
- "wmo08" or "WMO08", taken from [wmo08].
- "its90" or "ITS90", taken from [its90].

References

`xclim.indices._conversion.sfcwind_2_uas_vas(`*sfcWind: xr.DataArray*, *sfcWindfromdir: xr.DataArray*`)` → tuple[xr.DataArray, xr.DataArray]

Eastward and northward wind components from the wind speed and direction.

Compute the eastward and northward wind components from the wind speed and direction.

Parameters

- **sfcWind** (*xr.DataArray*) – Wind velocity
- **sfcWindfromdir** (*xr.DataArray*) – Direction from which the wind blows, following the meteorological convention where 360 stands for North.

Returns

- **uas** (*xr.DataArray, [m s-1]*) – Eastward wind velocity.
- **vas** (*xr.DataArray, [m s-1]*) – Northward wind velocity.

`xclim.indices._conversion.snowfall_approximation(`*pr: DataArray*, *tas: DataArray*, *thresh: str = '0 degC'*, *method: str = 'binary'*`)` → DataArray

Snowfall approximation from total precipitation and temperature.

Solid precipitation estimated from precipitation and temperature according to a given method.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **tas** (*xarray.DataArray, optional*) – Mean, maximum, or minimum daily temperature.
- **thresh** (*str,*) – Threshold temperature, used by method "binary".
- **method** (*{"binary", "brown", "auer"}*) – Which method to use when approximating snowfall from total precipitation. See notes.

Returns

xarray.DataArray, [same units as pr] – Solid precipitation flux.

Notes

The following methods are available to approximate snowfall and are drawn from the Canadian Land Surface Scheme (CLASS, [Verseghy09]).

- `'binary'` : When the temperature is under the freezing threshold, precipitation is assumed to be solid. The method is agnostic to the type of temperature used (mean, maximum or minimum).
- `'brown'` : The phase between the freezing threshold goes from solid to liquid linearly over a range of 2°C over the freezing point.
- `'auer'` : The phase between the freezing threshold goes from solid to liquid as a degree six polynomial over a range of 6°C over the freezing point.

References

https://gitlab.com/cccma/classic/-/blob/master/src/atmosphericVarsCalc.f90

`xclim.indices._conversion.specific_humidity(`*tas: DataArray*, *hurs: DataArray*, *ps: DataArray*, *ice_thresh: Optional[str] = None*, *method: str = 'sonntag90'*, *invalid_values: Optional[str] = None*) → DataArray

Specific humidity from temperature, relative humidity and pressure.

Specific humidity is the ratio between the mass of water vapour and the mass of moist air [WMO08].

Parameters

- **tas** (*xr.DataArray*) – Temperature array
- **hurs** (*xr.DataArray*) – Relative Humidity.
- **ps** (*xr.DataArray*) – Air Pressure.
- **ice_thresh** (*str*) – Threshold temperature under which to switch to equations in reference to ice instead of water. If None (default) everything is computed with reference to water.
- **method** (*{"goffgratch46", "sonntag90", "tetens30", "wmo08"}*) – Which method to use, see notes of this function and of *saturation_vapor_pressure*.
- **invalid_values** (*{"clip", "mask", None}*) – What to do with values larger than the saturation specific humidity and lower than 0. If "clip" (default), clips everything to 0 - q_sat if "mask", replaces values outside the range by np.nan, if None, does nothing.

Returns

xarray.DataArray, [dimensionless] – Specific humidity.

Notes

In the following, let T, $hurs$ (in %) and p be the temperature, the relative humidity and the air pressure. With w, w_{sat}, e_{sat} the mixing ratio, the saturation mixing ratio and the saturation vapor pressure, specific humidity q is computed as:

$$w_{sat} = 0.622\frac{e_{sat}}{P - e_{sat}} w = w_{sat} * hurs/100 q = w/(1 + w)$$

The methods differ by how e_{sat} is computed. See the doc of *xclim.core.utils.saturation_vapor_pressure*.

If *invalid_values* is not *None*, the saturation specific humidity q_{sat} is computed as:

$$q_{sat} = w_{sat}/(1 + w_{sat})$$

References

`xclim.indices._conversion.specific_humidity_from_dewpoint(`*tdps: DataArray*, *ps: DataArray*, *method: str = 'sonntag90'*`)` → DataArray

Specific humidity from dewpoint temperature and air pressure.

Specific humidity is the ratio between the mass of water vapour and the mass of moist air [WMO08].

Parameters

- **tdps** (*xr.DataArray*) – Dewpoint temperature array.
- **ps** (*xr.DataArray*) – Air pressure array.
- **method** (*{"goffgratch46", "sonntag90", "tetens30", "wmo08"}*) – Method to compute the saturation vapor pressure.

Returns

xarray.DataArray, [dimensionless] – Specific humidity.

Notes

If e is the water vapor pressure, and p the total air pressure, then specific humidity is given by

$$q = m_w e/(m_a(p - e) + m_w e)$$

where m_w and m_a are the molecular weights of water and dry air respectively. This formula is often written with $= m_w/m_a$, which simplifies to $q = e/(p - e(1-))$.

References

`xclim.indices._conversion.tas(`*tasmin: DataArray*, *tasmax: DataArray*`)` → DataArray

Average temperature from minimum and maximum temperatures.

We assume a symmetrical distribution for the temperature and retrieve the average value as Tg = (Tx + Tn) / 2

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum (daily) temperature
- **tasmax** (*xarray.DataArray*) – Maximum (daily) temperature

Returns

xarray.DataArray – Mean (daily) temperature [same units as tasmin]

`xclim.indices._conversion.uas_vas_2_sfcwind(`*uas: xr.DataArray*, *vas: xr.DataArray*, *calm_wind_thresh: str = '0.5 m/s'*`)` → tuple[xr.DataArray, xr.DataArray]

Wind speed and direction from the eastward and northward wind components.

Computes the magnitude and angle of the wind vector from its northward and eastward components, following the meteorological convention that sets calm wind to a direction of 0° and northerly wind to 360°.

Parameters

- **uas** (*xr.DataArray*) – Eastward wind velocity
- **vas** (*xr.DataArray*) – Northward wind velocity
- **calm_wind_thresh** (*str*) – The threshold under which winds are considered "calm" and for which the direction is set to 0. On the Beaufort scale, calm winds are defined as $<$ 0.5 m/s.

Returns

- **wind** (*xr.DataArray, [m s-1]*) – Wind velocity
- **wind_from_dir** (*xr.DataArray, [°]*) – Direction from which the wind blows, following the meteorological convention where 360 stands for North and 0 for calm winds.

Notes

Winds with a velocity less than *calm_wind_thresh* are given a wind direction of 0°, while stronger northerly winds are set to 360°.

`xclim.indices._conversion.universal_thermal_climate_index(`*tas: DataArray*, *hurs: DataArray*, *sfcWind: DataArray*, *mrt: Optional[DataArray] = None*, *rsds: Optional[DataArray] = None*, *rsus: Optional[DataArray] = None*, *rlds: Optional[DataArray] = None*, *rlus: Optional[DataArray] = None*, *stat: str = 'average'*, *mask_invalid: bool = True*`)` → DataArray

Universal thermal climate index.

The UTCI is the equivalent temperature for the environment derived from a reference environment and is used to evaluate heat stress in outdoor spaces.

Parameters

- **tas** (*xarray.DataArray*) – Mean temperature
- **hurs** (*xarray.DataArray*) – Relative Humidity
- **sfcWind** (*xarray.DataArray*) – Wind velocity
- **mrt** (*xarray.DataArray, optional*) – Mean radiant temperature
- **rsds** (*xr.DataArray, optional*) – Surface Downwelling Shortwave Radiation This is necessary if mrt is not None.

- **rsus** (*xr.DataArray, optional*) – Surface Upwelling Shortwave Radiation This is necessary if mrt is not None.
- **rlds** (*xr.DataArray, optional*) – Surface Downwelling Longwave Radiation This is necessary if mrt is not None.
- **rlus** (*xr.DataArray, optional*) – Surface Upwelling Longwave Radiation This is necessary if mrt is not None.
- **stat** (*{'average', 'instant', 'sunlit'}*) – Which statistic to apply. If "average", the average of the cosine of the solar zenith angle is calculated. If "instant", the instantaneous cosine of the solar zenith angle is calculated. If "sunlit", the cosine of the solar zenith angle is calculated during the sunlit period of each interval. If "instant", the instantaneous cosine of the solar zenith angle is calculated. This is necessary if mrt is not None.
- **mask_invalid** (*boolean*) – If True (default), UTCI values are NaN where any of the inputs are outside their validity ranges : -50°C < tas < 50°C, -30°C < tas - mrt < 30°C and 0.5 m/s < sfcWind < 17.0 m/s.

Returns

xarray.DataArray – Universal Thermal Climate Index.

Notes

The calculation uses water vapor partial pressure, which is derived from relative humidity and saturation vapor pressure computed according to the ITS-90 equation.

This code was inspired by the *pythermalcomfort* and *thermofeel* packages.

References

Bröde, Peter (2009). Program for calculating UTCI Temperature (UTCI), version a 0.002, http://www.utci.org/public/UTCI%20Program%20Code/UTCI_a002.f90 Błażejczyk, K., Jendritzky, G., Bröde, P., Fiala, D., Havenith, G., Epstein, Y., Psikuta, A., & Kampmann, B. (2013). An introduction to the Universal Thermal Climate Index (UTCI). DOI:10.7163/GPOL.2013.1

See also:

`http`

//www.utci.org/utcineu/utcineu.php

`xclim.indices._conversion.wind_chill_index`(*tas: DataArray*, *sfcWind: DataArray*, *method: str = 'CAN'*, *mask_invalid: bool = True*)

Wind chill index.

The Wind Chill Index is an estimation of how cold the weather feels to the average person. It is computed from the air temperature and the 10-m wind. As defined by the Environment and Climate Change Canada ([MVSZ2015]), two equations exist, the conventional one and one for slow winds (usually < 5 km/h), see Notes.

Parameters

- **tas** (*xarray.DataArray*) – Surface air temperature.
- **sfcWind** (*xarray.DataArray*) – Surface wind speed (10 m).

- **method** (*{'CAN', 'US'}*) – If "CAN" (default), a "slow wind" equation is used where winds are slower than 5 km/h, see Notes.
- **mask_invalid** (*bool*) – Whether to mask values when the inputs are outside their validity range. or not. If True (default), points where the temperature is above a threshold are masked. The threshold is 0°C for the canadian method and 50°F for the american one. With the latter method, points where sfcWind < 3 mph are also masked.

Returns
xarray.DataArray, [degC] – Wind Chill Index.

Notes

Following the calculations of Environment and Climate Change Canada, this function switches from the standardized index to another one for slow winds. The standard index is the same as used by the National Weather Service of the USA ([NWS]). Given a temperature at surface T (in °C) and 10-m wind speed V (in km/h), the Wind Chill Index W (dimensionless) is computed as:

$$W = 13.12 + 0.6125 * T - 11.37 * V^0.16 + 0.3965 * T * V^0.16$$

Under slow winds ($V < 5$ km/h), and using the canadian method, it becomes:

$$W = T + \frac{-1.59 + 0.1345 * T}{5} * V$$

Both equations are invalid for temperature over 0°C in the canadian method.

The american Wind Chill Temperature index (WCT), as defined by USA's National Weather Service, is computed when *method='US'*. In that case, the maximal valid temperature is 50°F (10 °C) and minimal wind speed is 3 mph (4.8 km/h).

See also:

`National`

References

xclim.indices._hydrology module

`xclim.indices._hydrology.base_flow_index(`*q: DataArray, freq: str = 'YS'*`)` → DataArray

Base flow index.

Return the base flow index, defined as the minimum 7-day average flow divided by the mean flow.

Parameters

- **q** (*xarray.DataArray*) – Rate of river discharge.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [dimensionless] – Base flow index.

Notes

Let $\mathbf{q} = q_0, q_1, \ldots, q_n$ be the sequence of daily discharge and $\overline{\mathbf{q}}$ the mean flow over the period. The base flow index is given by:

$$\frac{\min(\mathrm{CMA}_7(\mathbf{q}))}{\overline{\mathbf{q}}}$$

where CMA_7 is the seven days moving average of the daily flow:

$$\mathrm{CMA}_7(q_i) = \frac{\sum_{j=i-3}^{i+3} q_j}{7}$$

`xclim.indices._hydrology.melt_and_precip_max(`*snw: DataArray*, *pr: DataArray*, *window: int = 3*, *freq: str = 'AS-JUL'*`)` → DataArray

Maximum snow melt and precipitation.

The maximum snow melt plus precipitation over a given number of days expressed in snow water equivalent.

Parameters

- **snw** (*xarray.DataArray*) – Snow amount (mass per area).
- **pr** (*xarray.DataArray*) – Daily precipitation flux.
- **window** (*int*) – Number of days during which the water input is accumulated.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – The maximum snow melt plus precipitation over a given number of days for each period. [mass/area].

`xclim.indices._hydrology.rb_flashiness_index(`*q: DataArray*, *freq: str = 'YS'*`)` → DataArray

Richards-Baker flashiness index.

Measures oscillations in flow relative to total flow, quantifying the frequency and rapidity of short term changes in flow, based on Baker et al. (2004; [baker2004]).

Parameters

- **q** (*xarray.DataArray*) – Rate of river discharge.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – R-B Index.

Notes

Let $\mathbf{q} = q_0, q_1, \ldots, q_n$ be the sequence of daily discharge, the R-B Index is given by:

$$\frac{\sum_{i=1}^{n} |q_i - q_{i-1}|}{\sum_{i=1}^{n} q_i}$$

References

`xclim.indices._hydrology.snd_max_doy(`*snd: DataArray*, *freq: str = 'AS-JUL'*`)` → DataArray

Maximum snow depth day of year.

Day of year when surface snow reaches its peak value. If snow depth is 0 over entire period, return NaN.

Parameters

- **snd** (*xarray.DataArray*) – Surface snow depth.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – The day of year at which snow depth reaches its maximum value.

`xclim.indices._hydrology.snow_melt_we_max(`*snw: DataArray*, *window: int = 3*, *freq: str = 'AS-JUL'*`)` → DataArray

Maximum snow melt.

The maximum snow melt over a given number of days expressed in snow water equivalent.

Parameters

- **snw** (*xarray.DataArray*) – Snow amount (mass per area).
- **window** (*int*) – Number of days during which the melt is accumulated.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – The maximum snow melt over a given number of days for each period. [mass/area].

`xclim.indices._hydrology.snw_max(`*snw: DataArray*, *freq: str = 'AS-JUL'*`)` → DataArray

Maximum snow amount.

The maximum daily snow amount.

Parameters

- **snw** (*xarray.DataArray*) – Snow amount (mass per area).
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – The maximum snow amount over a given number of days for each period. [mass/area].

`xclim.indices._hydrology.snw_max_doy(`*snw: DataArray*, *freq: str = 'AS-JUL'*`)` → DataArray

Maximum snow amount day of year.

Day of year when surface snow amount reaches its peak value. If snow amount is 0 over entire period, return NaN.

Parameters

- **snw** (*xarray.DataArray*) – Surface snow amount.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – The day of year at which snow amount reaches its maximum value.

xclim.indices._multivariate module

`xclim.indices._multivariate.blowing_snow(`*snd: DataArray, sfcWind: DataArray, snd_thresh: str = '5 cm', sfcWind_thresh: str = '15 km/h', window: int = 3, freq: str = 'AS-JUL'*`)` → DataArray

Days with blowing snow events.

Number of days where both snowfall over the last days and daily wind speeds are above respective thresholds.

Parameters

- **snd** (*xarray.DataArray*) – Surface snow depth.
- **sfcWind** (*xr.DataArray*) – Wind velocity
- **snd_thresh** (*str*) – Threshold on net snowfall accumulation over the last *window* days.
- **sfcWind_thresh** (*str*) – Wind speed threshold.
- **window** (*int*) – Period over which snow is accumulated before comparing against threshold.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – Number of days where snowfall and wind speeds are above respective thresholds.

`xclim.indices._multivariate.cold_and_dry_days(`*tas: DataArray, pr: DataArray, tas_per: DataArray, pr_per: DataArray, freq: str = 'YS'*`)` → DataArray

Cold and dry days.

Returns the total number of days where “Cold” and “Dry” conditions coincide.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature values
- **pr** (*xarray.DataArray*) – Daily precipitation.
- **tas_per** (*xarray.DataArray*) – First quartile of daily mean temperature computed by month.
- **pr_per** (*xarray.DataArray*) – First quartile of daily total precipitation computed by month.

  Warning: Before computing the percentiles, all the precipitation below 1mm must be filtered out ! Otherwise, the percentiles will include non-wet days.

- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, – The total number of days where cold and dry conditions coincide.

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [cold_dry_days].

References

`xclim.indices._multivariate.cold_and_wet_days(`*tas: DataArray, pr: DataArray, tas_per: DataArray, pr_per: DataArray, freq: str = 'YS'*`)` → DataArray

Cold and wet days.

Returns the total number of days where "cold" and "wet" conditions coincide.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature values
- **pr** (*xarray.DataArray*) – Daily precipitation.
- **tas_per** (*xarray.DataArray*) – First quartile of daily mean temperature computed by month.
- **pr_per** (*xarray.DataArray*) – Third quartile of daily total precipitation computed by month.
- **freq** (*str*) – Resampling frequency.

Warning: Before computing the percentiles, all the precipitation below 1mm must be filtered out! Otherwise, the percentiles will include non-wet days.

Returns

xarray.DataArray, – The total number of days where cold and wet conditions coincide.

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [cold_wet_days].

References

`xclim.indices._multivariate.cold_spell_duration_index(`*tasmin: DataArray, tasmin_per: DataArray, window: int = 6, freq: str = 'YS', bootstrap: bool = False*`)` → DataArray

Cold spell duration index.

Number of days with at least *window* consecutive days where the daily minimum temperature is below the *tasmin_per* percentiles.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmin_per** (*xarray.DataArray*) – nth percentile of daily minimum temperature with *dayofyear* coordinate.
- **window** (*int*) – Minimum number of days with temperature below threshold to qualify as a cold spell.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [time] – Count of days with at least six consecutive days where the daily minimum temperature is below the 10th percentile.

Notes

Let TN_i be the minimum daily temperature for the day of the year i and $TN10_i$ the 10th percentile of the minimum daily temperature over the 1961-1990 period for day of the year i, the cold spell duration index over period ϕ is defined as:

$$\sum_{i\in\phi}\prod_{j=i}^{i+6}[TN_j < TN10_j]$$

where $[P]$ is 1 if P is true, and 0 if false.

References

From the Expert Team on Climate Change Detection, Monitoring and Indices (ETCCDMI).

Examples

# Note that this example does not use a proper 1961-1990 reference period. >>> from xclim.core.calendar import percentile_doy >>> from xclim.indices import cold_spell_duration_index

```
>>> tasmin = xr.open_dataset(path_to_tasmin_file).tasmin.isel(lat=0, lon=0)
>>> tn10 = percentile_doy(tasmin, per=10).sel(percentiles=10)
>>> cold_spell_duration_index(tasmin, tn10)
```

`xclim.indices._multivariate.daily_freezethaw_cycles`(*tasmin: DataArray, tasmax: DataArray, thresh_tasmin: str = '0 degC', thresh_tasmax: str = '0 degC', freq: str = 'YS'*) → DataArray

Number of days with a diurnal freeze-thaw cycle.

The number of days where Tmax > thresh_tasmax and Tmin <= thresh_tasmin.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – The temperature threshold needed to trigger a freeze event.
- **thresh_tasmax** (*str*) – The temperature threshold needed to trigger a thaw event.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Number of days with a diurnal freeze-thaw cycle

Notes

Let TX_i be the maximum temperature at day i and TN_i be the daily minimum temperature at day i. Then the number of freeze thaw cycles during period ϕ is given by :

$$\sum_{i \in \phi} [TX_i > 0][TN_i < 0]$$

where $[P]$ is 1 if P is true, and 0 if false.

Warning: The *daily_freezethaw_cycles* indice is being deprecated in favour of *multiday_temperature_swing* with *thresh_tasmax='0 degC, thresh_tasmin='0 degC', window=1, op='sum'* by default. The indicator reflects this change. This indice will be removed in a future version of xclim.

`xclim.indices._multivariate.daily_temperature_range`(*tasmin: DataArray*, *tasmax: DataArray*, *freq: str = 'YS'*, *op: str = 'mean'*) → DataArray

Statistics of daily temperature range.

The mean difference between the daily maximum temperature and the daily minimum temperature.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **freq** (*str*) – Resampling frequency.
- **op** (*{'min', 'max', 'mean', 'std'} or func*) – Reduce operation. Can either be a DataArray method or a function that can be applied to a DataArray.

Returns

xarray.DataArray, [same units as tasmin] – The average variation in daily temperature range for the given time period.

Notes

For a default calculation using *op='mean'* :

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the mean diurnal temperature range in period j is:

$$DTR_j = \frac{\sum_{i=1}^{I}(TX_{ij} - TN_{ij})}{I}$$

`xclim.indices._multivariate.daily_temperature_range_variability(`*tasmin: DataArray*, *tasmax: DataArray*, *freq: str = 'YS'*`)` → DataArray

Mean absolute day-to-day variation in daily temperature range.

Mean absolute day-to-day variation in daily temperature range.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tasmin] – The average day-to-day variation in daily temperature range for the given time period.

Notes

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then calculated is the absolute day-to-day differences in period j is:

$$vDTR_j = \frac{\sum_{i=2}^{I}|(TX_{ij} - TN_{ij}) - (TX_{i-1,j} - TN_{i-1,j})|}{I}$$

`xclim.indices._multivariate.days_over_precip_thresh(`*pr: DataArray*, *pr_per: DataArray*, *thresh: str = '1 mm/day'*, *freq: str = 'YS'*, *bootstrap: bool = False*`)` → DataArray

Number of wet days with daily precipitation over a given percentile.

Number of days over period where the precipitation is above a threshold defining wet days and above a given percentile for that day.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **pr_per** (*xarray.DataArray*) – Percentile of wet day precipitation flux. Either computed daily (one value per day of year) or computed over a period (one value per spatial point).
- **thresh** (*str*) – Precipitation value over which a day is considered wet.
- **freq** (*str*) – Resampling frequency.

- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns
xarray.DataArray, [time] – Count of days with daily precipitation above the given percentile [days].

Examples

```
>>> from xclim.indices import days_over_precip_thresh
>>> pr = xr.open_dataset(path_to_pr_file).pr
>>> p75 = pr.quantile(0.75, dim="time", keep_attrs=True)
>>> r75p = days_over_precip_thresh(pr, p75)
```

`xclim.indices._multivariate.extreme_temperature_range(`*tasmin: DataArray*, *tasmax: DataArray*, *freq: str = 'YS'*) → DataArray

Extreme intra-period temperature range.

The maximum of max temperature (TXx) minus the minimum of min temperature (TNn) for the given time period.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [same units as tasmin] – Extreme intra-period temperature range for the given time period.

Notes

Let TX_{ij} and TN_{ij} be the daily maximum and minimum temperature at day i of period j. Then the extreme temperature range in period j is:

$$ETR_j = max(TX_{ij}) - min(TN_{ij})$$

`xclim.indices._multivariate.fraction_over_precip_thresh(`*pr: DataArray*, *pr_per: DataArray*, *thresh: str = '1 mm/day'*, *freq: str = 'YS'*, *bootstrap: bool = False*) → DataArray

Fraction of precipitation due to wet days with daily precipitation over a given percentile.

Percentage of the total precipitation over period occurring in days where the precipitation is above a threshold defining wet days and above a given percentile for that day.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.

- **pr_per** (*xarray.DataArray*) – Percentile of wet day precipitation flux. Either computed daily (one value per day of year) or computed over a period (one value per spatial point).
- **thresh** (*str*) – Precipitation value over which a day is considered wet.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns
xarray.DataArray, [dimensionless] – Fraction of precipitation over threshold during wet days.

`xclim.indices._multivariate.heat_wave_frequency(`*tasmin: DataArray, tasmax: DataArray, thresh_tasmin: str = '22.0 degC', thresh_tasmax: str = '30 degC', window: int = 3, freq: str = 'YS'*`)` → DataArray

Heat wave frequency.

Number of heat waves over a given period. A heat wave is defined as an event where the minimum and maximum daily temperature both exceeds specific thresholds over a minimum number of days.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – The minimum temperature threshold needed to trigger a heatwave event.
- **thresh_tasmax** (*str*) – The maximum temperature threshold needed to trigger a heatwave event.
- **window** (*int*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [dimensionless] – Number of heatwave at the requested frequency.

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities ([casati2013]).

In Robinson (2001; [robinson2001]), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

`xclim.indices._multivariate.heat_wave_max_length(`*tasmin: DataArray*, *tasmax: DataArray*, *thresh_tasmin: str = '22.0 degC'*, *thresh_tasmax: str = '30 degC'*, *window: int = 3*, *freq: str = 'YS'*`)` → DataArray

Heat wave max length.

Maximum length of heat waves over a given period. A heat wave is defined as an event where the minimum and maximum daily temperature both exceeds specific thresholds over a minimum number of days.

By definition heat_wave_max_length must be >= window.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – The minimum temperature threshold needed to trigger a heatwave event.
- **thresh_tasmax** (*str*) – The maximum temperature threshold needed to trigger a heatwave event.
- **window** (*int*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Maximum length of heatwave at the requested frequency.

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities ([casati2013]).

In Robinson (2001; [robinson2001]), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

`xclim.indices._multivariate.heat_wave_total_length(`*tasmin: DataArray*, *tasmax: DataArray*, *thresh_tasmin: str = '22.0 degC'*, *thresh_tasmax: str = '30 degC'*, *window: int = 3*, *freq: str = 'YS'*`)` → DataArray

Heat wave total length.

Total length of heat waves over a given period. A heat wave is defined as an event where the minimum and maximum daily temperature both exceeds specific thresholds over a minimum number of days. This the sum of all days in such events.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – The minimum temperature threshold needed to trigger a heatwave event.
- **thresh_tasmax** (*str*) – The maximum temperature threshold needed to trigger a heatwave event.
- **window** (*int*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Total length of heatwave at the requested frequency.

Notes

See notes and references of *heat_wave_max_length*

`xclim.indices._multivariate.high_precip_low_temp(`*pr: DataArray, tas: DataArray, pr_thresh: str = '0.4 mm/d', tas_thresh: str = '-0.2 degC', freq: str = 'YS'*`)` → DataArray

Number of days with precipitation above threshold and temperature below threshold.

Number of days where precipitation is greater or equal to some threshold, and temperatures are colder than some threshold. This can be used for example to identify days with the potential for freezing rain or icing conditions.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **tas** (*xarray.DataArray*) – Daily mean, minimum or maximum temperature.
- **pr_thresh** (*str*) – Precipitation threshold to exceed.
- **tas_thresh** (*str*) – Temperature threshold not to exceed.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Count of days with high precipitation and low temperatures.

Example

To compute the number of days with intense rainfall while minimum temperatures dip below -0.2C: >>> pr = xr.open_dataset(path_to_pr_file).pr >>> tasmin = xr.open_dataset(path_to_tasmin_file).tasmin >>> high_precip_low_temp(… pr, tas=tasmin, pr_thresh="10 mm/d", tas_thresh="-0.2 degC" …)

`xclim.indices._multivariate.liquid_precip_ratio(`*pr: xarray.DataArray, prsn: xarray.DataArray | None = None, tas: xarray.DataArray | None = None, thresh: str = '0 degC', freq: str = 'QS-DEC'*`)` → xarray.DataArray

Ratio of rainfall to total precipitation.

The ratio of total liquid precipitation over the total precipitation. If solid precipitation is not provided, it is approximated with pr, tas and thresh, using the *snowfall_approximation* function with method 'binary'.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **prsn** (*xarray.DataArray, optional*) – Mean daily solid precipitation flux.
- **tas** (*xarray.DataArray, optional*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature under which precipitation is assumed to be solid.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Ratio of rainfall to total precipitation.

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$
$$PRwet_{ij}$$

See also:

winter_rain_ratio

`xclim.indices._multivariate.multiday_temperature_swing(`*tasmin: DataArray, tasmax: DataArray, thresh_tasmin: str = '0 degC', thresh_tasmax: str = '0 degC', window: int = 1, op: str = 'mean', freq: str = 'YS'*`)` → DataArray

Statistics of consecutive diurnal temperature swing events.

A diurnal swing of max and min temperature event is when Tmax > thresh_tasmax and Tmin <= thresh_tasmin. This indice finds all days that constitute these events and computes statistics over the length and frequency of these events.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – The temperature threshold needed to trigger a freeze event.
- **thresh_tasmax** (*str*) – The temperature threshold needed to trigger a thaw event.
- **window** (*int*) – The minimal length of spells to be included in the statistics.
- **op** (*{'mean', 'sum', 'max', 'min', 'std', 'count'}*) – The statistical operation to use when reducing the list of spell lengths.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – {freq} {op} length of diurnal temperature cycles exceeding thresholds.

Notes

Let TX_i be the maximum temperature at day i and TN_i be the daily minimum temperature at day i. Then freeze thaw spells during a given period are consecutive days where:

$$TX_i > 0 \land TN_i < 0$$

This indice returns a given statistic of the found lengths, optionally dropping those shorter than the *window* argument. For example, *window=1* and *op='sum'* returns the same value as `daily_freezethaw_cycles()`.

`xclim.indices._multivariate.precip_accumulation(`*pr: xarray.DataArray*, *tas: xarray.DataArray = None*, *phase: str | None = None*, *thresh: str = '0 degC'*, *freq: str = 'YS'*`)` → xarray.DataArray

Accumulated total (liquid and/or solid) precipitation.

Resample the original daily mean precipitation flux and accumulate over each period. If a daily temperature is provided, the *phase* keyword can be used to sum precipitation of a given phase only. When the temperature is under the provided threshold, precipitation is assumed to be snow, and liquid rain otherwise. This indice is agnostic to the type of daily temperature (tas, tasmax or tasmin) given.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **tas** (*xarray.DataArray, optional*) – Mean, maximum or minimum daily temperature.
- **phase** (*{None, 'liquid', 'solid'}*) – Which phase to consider, "liquid" or "solid", if None (default), both are considered.
- **thresh** (*str*) – Threshold of *tas* over which the precipication is assumed to be liquid rain.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [length] – The total daily precipitation at the given time frequency for the given phase.

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j starting at day a and finishing on day b:

$$PR_{ij} = \sum_{i=a}^{b} PR_i$$

If tas and phase are given, the corresponding phase precipitation is estimated before computing the accumulation, using one of *snowfall_approximation* or *rain_approximation* with the *binary* method.

Examples

The following would compute, for each grid cell of a dataset, the total precipitation at the seasonal frequency, ie DJF, MAM, JJA, SON, DJF, etc.:

```
>>> from xclim.indices import precip_accumulation
>>> pr_day = xr.open_dataset(path_to_pr_file).pr
>>> prcp_tot_seasonal = precip_accumulation(pr_day, freq="QS-DEC")
```

`xclim.indices._multivariate.rain_on_frozen_ground_days`(*pr: DataArray*, *tas: DataArray*, *thresh: str = '1 mm/d'*, *freq: str = 'YS'*) → DataArray

Number of rain on frozen ground events.

Number of days with rain above a threshold after a series of seven days below freezing temperature. Precipitation is assumed to be rain when the temperature is above 0℃.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Precipitation threshold to consider a day as a rain event.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – The number of rain on frozen ground events per period.

Notes

Let PR_i be the mean daily precipitation and TG_i be the mean daily temperature of day i. Then for a period j, rain on frozen grounds days are counted where:

$$PR_i > Threshold[mm]$$

and where

$$TG_i 0$$

is true for continuous periods where $i 7$

`xclim.indices._multivariate.tg10p`(*tas: DataArray*, *tas_per: DataArray*, *freq: str = 'YS'*, *bootstrap: bool = False*) → DataArray

Number of days with daily mean temperature below the 10th percentile.

Number of days with daily mean temperature below the 10th percentile.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **tas_per** (*xarray.DataArray*) – 10th percentile of daily mean temperature.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and

the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [time] – Count of days with daily mean temperature below the 10th percentile [days].

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tg10p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> tas_per = percentile_doy(tas, per=10).sel(percentiles=10)
>>> cold_days = tg10p(tas, tas_per)
```

`xclim.indices._multivariate.tg90p(`*tas: DataArray*, *tas_per: DataArray*, *freq: str = 'YS'*, *bootstrap: bool = False*`)` → DataArray

Number of days with daily mean temperature over the 90th percentile.

Number of days with daily mean temperature over the 90th percentile.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **tas_per** (*xarray.DataArray*) – 90th percentile of daily mean temperature.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [time] – Count of days with daily mean temperature below the 10th percentile [days].

Notes

The 90th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tg90p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> tas_per = percentile_doy(tas, per=90).sel(percentiles=90)
>>> hot_days = tg90p(tas, tas_per)
```

`xclim.indices._multivariate.tn10p`(*tasmin: DataArray*, *tasmin_per: DataArray*, *freq: str = 'YS'*, *bootstrap: bool = False*) → DataArray

Number of days with daily minimum temperature below the 10th percentile.

Number of days with daily minimum temperature below the 10th percentile.

Parameters

- **tasmin** (*xarray.DataArray*) – Mean daily temperature.
- **tasmin_per** (*xarray.DataArray*) – 10th percentile of daily minimum temperature.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [time] – Count of days with daily minimum temperature below the 10th percentile [days].

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tn10p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> tas_per = percentile_doy(tas, per=10).sel(percentiles=10)
>>> cold_days = tn10p(tas, tas_per)
```

`xclim.indices._multivariate.tn90p`(*tasmin: DataArray*, *tasmin_per: DataArray*, *freq: str = 'YS'*, *bootstrap: bool = False*) → DataArray

Number of days with daily minimum temperature over the 90th percentile.

Number of days with daily minimum temperature over the 90th percentile.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmin_per** (*xarray.DataArray*) – 90th percentile of daily minimum temperature.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [time] – Count of days with daily minimum temperature below the 10th percentile [days].

Notes

The 90th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tn90p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> tas_per = percentile_doy(tas, per=90).sel(percentiles=90)
>>> hot_days = tn90p(tas, tas_per)
```

`xclim.indices._multivariate.tx10p`(*tasmax: DataArray*, *tasmax_per: DataArray*, *freq: str = 'YS'*, *bootstrap: bool = False*) → DataArray

Number of days with daily maximum temperature below the 10th percentile.

Number of days with daily maximum temperature below the 10th percentile.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **tasmax_per** (*xarray.DataArray*) – 10th percentile of daily maximum temperature.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns
xarray.DataArray, [time] – Count of days with daily maximum temperature below the 10th percentile [days].

Notes

The 10th percentile should be computed for a 5 day window centered on each calendar day for a reference period.

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tx10p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> tasmax_per = percentile_doy(tas, per=10).sel(percentiles=10)
>>> cold_days = tx10p(tas, tasmax_per)
```

`xclim.indices._multivariate.tx90p(`*tasmax: DataArray, tasmax_per: DataArray, freq: str = 'YS', bootstrap: bool = False*`)` → DataArray

Number of days with daily maximum temperature over the 90th percentile.

Number of days with daily maximum temperature over the 90th percentile.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **tasmax_per** (*xarray.DataArray*) – 90th percentile of daily maximum temperature.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns
xarray.DataArray, [time] – Count of days with daily maximum temperature below the 10th percentile [days].

Notes

The 90th percentile should be computed for a 5-day window centered on each calendar day for a reference period.

Examples

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import tx90p
>>> tas = xr.open_dataset(path_to_tas_file).tas
>>> tasmax_per = percentile_doy(tas, per=90).sel(percentiles=90)
>>> hot_days = tx90p(tas, tasmax_per)
```

`xclim.indices._multivariate.tx_tn_days_above`(*tasmin: DataArray*, *tasmax: DataArray*, *thresh_tasmin: str = '22 degC'*, *thresh_tasmax: str = '30 degC'*, *freq: str = 'YS'*) → DataArray

Number of days with both hot maximum and minimum daily temperatures.

The number of days per period with tasmin above a threshold and tasmax above another threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmin** (*str*) – Threshold temperature for tasmin on which to base evaluation.
- **thresh_tasmax** (*str*) – Threshold temperature for tasmax on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – the number of days with tasmin > thresh_tasmin and tasmax > thresh_tasmax per period.

Notes

Let TX_{ij} be the maximum temperature at day i of period j, TN_{ij} the daily minimum temperature at day i of period j, TX_{thresh} the threshold for maximum daily temperature, and TN_{thresh} the threshold for minimum daily temperature. Then counted is the number of days where:

$$TX_{ij} > TX_{thresh}[]$$

and where:

$$TN_{ij} > TN_{thresh}[]$$

`xclim.indices._multivariate.warm_and_dry_days`(*tas: DataArray*, *pr: DataArray*, *tas_per: DataArray*, *pr_per: DataArray*, *freq: str = 'YS'*) → DataArray

Warm and dry days.

Returns the total number of days where "warm" and "Dry" conditions coincide.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature values
- **pr** (*xarray.DataArray*) – Daily precipitation.

- **tas_per** (*xarray.DataArray*) – Third quartile of daily mean temperature computed by month.
- **pr_per** (*xarray.DataArray*) – First quartile of daily total precipitation computed by month.

> **Warning:** Before computing the percentiles, all the precipitation below 1mm must be filtered out ! Otherwise, the percentiles will include non-wet days.

- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, – The total number of days where warm and dry conditions coincide.

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [warm_dry_days].

References

`xclim.indices._multivariate.warm_and_wet_days(`*tas: DataArray, pr: DataArray, tas_per: DataArray, pr_per: DataArray, freq: str = 'YS'*`)` → DataArray

Warm and wet days.

Returns the total number of days where "warm" and "wet" conditions coincide.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature values
- **pr** (*xarray.DataArray*) – Daily precipitation.
- **tas_per** (*xarray.DataArray*) – Third quartile of daily mean temperature computed by month.
- **pr_per** (*xarray.DataArray*) – Third quartile of daily total precipitation computed by month.

> **Warning:** Before computing the percentiles, all the precipitation below 1mm must be filtered out ! Otherwise, the percentiles will include non-wet days.

- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, – The total number of days where warm and wet conditions coincide.

Notes

Bootstrapping is not available for quartiles because it would make no significant difference to bootstrap percentiles so far from the extremes.

Formula to be written [warm_wet_days].

References

`xclim.indices._multivariate.warm_spell_duration_index(`*tasmax: DataArray*, *tasmax_per: DataArray*, *window: int = 6*, *freq: str = 'YS'*, *bootstrap: bool = False*`)` → DataArray

Warm spell duration index.

Number of days inside spells of a minimum number of consecutive days where the daily maximum temperature is above the 90th percentile. The 90th percentile should be computed for a 5-day moving window, centered on each calendar day in the 1961-1990 period.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **tasmax_per** (*xarray.DataArray*) – percentile(s) of daily maximum temperature.
- **window** (*int*) – Minimum number of days with temperature above threshold to qualify as a warm spell.
- **freq** (*str*) – Resampling frequency.
- **bootstrap** (*bool*) – Flag to run bootstrapping of percentiles. Used by percentile_bootstrap decorator. Bootstrapping is only useful when the percentiles are computed on a part of the studied sample. This period, common to percentiles and the sample must be bootstrapped to avoid inhomogeneities with the rest of the time series. Keep bootstrap to False when there is no common period, it would give wrong results plus, bootstrapping is computationally expensive.

Returns

xarray.DataArray, [time] – Warm spell duration index.

References

From the Expert Team on Climate Change Detection, Monitoring and Indices (ETCCDMI). Used in Alexander, L. V., et al. (2006), Global observed changes in daily climate extremes of temperature and precipitation, J. Geophys. Res., 111, D05109, doi: 10.1029/2005JD006290.

Examples

Note that this example does not use a proper 1961-1990 reference period.

```
>>> from xclim.core.calendar import percentile_doy
>>> from xclim.indices import warm_spell_duration_index
```

```
>>> tasmax = xr.open_dataset(path_to_tasmax_file).tasmax.isel(lat=0, lon=0)
>>> tasmax_per = percentile_doy(tasmax, per=90).sel(percentiles=90)
>>> warm_spell_duration_index(tasmax, tasmax_per)
```

`xclim.indices._multivariate.winter_rain_ratio(`**, pr: DataArray, prsn: Optional[DataArray] = None, tas: Optional[DataArray] = None, freq: str = 'QS-DEC'*`)` → DataArray

Ratio of rainfall to total precipitation during winter.

The ratio of total liquid precipitation over the total precipitation over the winter months (DJF. If solid precipitation is not provided, then precipitation is assumed solid if the temperature is below 0°C.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **prsn** (*xarray.DataArray, optional*) – Mean daily solid precipitation flux.
- **tas** (*xarray.DataArray, optional*) – Mean daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – Ratio of rainfall to total precipitation during winter months (DJF).

xclim.indices._simple module

`xclim.indices._simple.frost_days(`*tasmin: DataArray, thresh: str = '0 degC', freq: str = 'YS'*`)` → DataArray

Frost days index.

Number of days where daily minimum temperatures are below a threshold temperature.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Freezing temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Frost days index.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j and :math`TT` the threshold. Then counted is the number of days where:

$$TN_{ij} < TT$$

`xclim.indices._simple.ice_days(`*tasmax: DataArray, thresh: str = '0 degC', freq: str = 'YS'*`)` → DataArray

Number of ice/freezing days.

Number of days where daily maximum temperatures are below a threshold.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh** (*str*) – Freezing temperature.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of ice/freezing days.

Notes

Let TX_{ij} be the daily maximum temperature at day i of period j, and :math`TT` the threshold. Then counted is the number of days where:

$$TX_{ij} < TT$$

`xclim.indices._simple.max_1day_precipitation_amount(`*pr: DataArray, freq: str = 'YS'*`)` → DataArray

Highest 1-day precipitation amount for a period (frequency).

Resample the original daily total precipitation temperature series by taking the max over each period.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation values.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [same units as pr] – The highest 1-period precipitation flux value at the given time frequency.

Notes

Let PR_i be the mean daily precipitation of day i, then for a period j:

$$PRx_{ij} = max(PR_{ij})$$

Examples

```
>>> from xclim.indices import max_1day_precipitation_amount
```

# The following would compute for each grid cell the highest 1-day total # at an annual frequency: >>> pr = xr.open_dataset(path_to_pr_file).pr >>> rx1day = max_1day_precipitation_amount(pr, freq="YS")

`xclim.indices._simple.max_n_day_precipitation_amount(`*pr: DataArray, window: int = 1, freq: str = 'YS'*`)` → DataArray

Highest precipitation amount cumulated over a n-day moving window.

Calculate the n-day rolling sum of the original daily total precipitation series and determine the maximum value over each period.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation values.
- **window** (*int*) – Window size in days.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [length] – The highest cumulated n-period precipitation value at the given time frequency.

Examples

```
>>> from xclim.indices import max_n_day_precipitation_amount
```

# The following would compute for each grid cell the highest 5-day total precipitation #at an annual frequency: >>> pr = xr.open_dataset(path_to_pr_file).pr >>> out = max_n_day_precipitation_amount(pr, window=5, freq="YS")

`xclim.indices._simple.max_pr_intensity`(*pr: DataArray*, *window: int = 1*, *freq: str = 'YS'*) → DataArray

Highest precipitation intensity over a n-hour moving window.

Calculate the n-hour rolling average of the original hourly total precipitation series and determine the maximum value over each period.

Parameters

- **pr** (*xarray.DataArray*) – Hourly precipitation values.
- **window** (*int*) – Window size in hours.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as pr] – The highest cumulated n-hour precipitation intensity at the given time frequency.

Examples

```
>>> from xclim.indices import max_pr_intensity
```

# The following would compute the maximum 6-hour precipitation intensity. # at an annual frequency: # TODO

`xclim.indices._simple.snow_depth`(*snd: DataArray*, *freq: str = 'YS'*) → DataArray

Mean of daily average snow depth.

Resample the original daily mean snow depth series by taking the mean over each period.

Parameters

- **snd** (*xarray.DataArray*) – Mean daily snow depth.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as snd] – The mean daily snow depth at the given time frequency

`xclim.indices._simple.tg_max`(*tas: DataArray*, *freq: str = 'YS'*) → DataArray

Highest mean temperature.

The maximum of daily mean temperature.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tas] – Maximum of daily minimum temperature.

Notes

Let TN_{ij} be the mean temperature at day i of period j. Then the maximum daily mean temperature for period j is:

$$TNx_j = max(TN_{ij})$$

`xclim.indices._simple.tg_mean(`*tas: DataArray, freq: str = 'YS'*`)` → DataArray

Mean of daily average temperature.

Resample the original daily mean temperature series by taking the mean over each period.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tas] – The mean daily temperature at the given time frequency

Notes

Let TN_i be the mean daily temperature of day i, then for a period p starting at day a and finishing on day b:

$$TG_p = \frac{\sum_{i=a}^{b} TN_i}{b - a + 1}$$

Examples

The following would compute for each grid cell of file *tas.day.nc* the mean temperature at the seasonal frequency, ie DJF, MAM, JJA, SON, DJF, etc.:

```
>>> from xclim.indices import tg_mean
>>> t = xr.open_dataset(path_to_tas_file).tas
>>> tg = tg_mean(t, freq="QS-DEC")
```

`xclim.indices._simple.tg_min(`*tas: DataArray, freq: str = 'YS'*`)` → DataArray

Lowest mean temperature.

Minimum of daily mean temperature.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tas] – Minimum of daily minimum temperature.

Notes

Let TG_{ij} be the mean temperature at day i of period j. Then the minimum daily mean temperature for period j is:

$$TGn_j = min(TG_{ij})$$

`xclim.indices._simple.tn_max(`*tasmin: DataArray*, *freq: str = 'YS'*`)` → DataArray

Highest minimum temperature.

The maximum of daily minimum temperature.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tasmin] – Maximum of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then the maximum daily minimum temperature for period j is:

$$TNx_j = max(TN_{ij})$$

`xclim.indices._simple.tn_mean(`*tasmin: DataArray*, *freq: str = 'YS'*`)` → DataArray

Mean minimum temperature.

Mean of daily minimum temperature.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tasmin] – Mean of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then mean values in period j are given by:

$$TN_{ij} = \frac{\sum_{i=1}^{I} TN_{ij}}{I}$$

`xclim.indices._simple.tn_min(`*tasmin: DataArray*, *freq: str = 'YS'*`)` → DataArray

Lowest minimum temperature.

Minimum of daily minimum temperature.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.

- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [same units as tasmin] – Minimum of daily minimum temperature.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then the minimum daily minimum temperature for period j is:

$$TNn_j = min(TN_{ij})$$

`xclim.indices._simple.tx_max(`*tasmax: DataArray*, *freq: str = 'YS'*`)` → DataArray

Highest max temperature.

The maximum value of daily maximum temperature.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [same units as tasmax] – Maximum value of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then the maximum daily maximum temperature for period j is:

$$TXx_j = max(TX_{ij})$$

`xclim.indices._simple.tx_mean(`*tasmax: DataArray*, *freq: str = 'YS'*`)` → DataArray

Mean max temperature.

The mean of daily maximum temperature.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [same units as tasmax] – Mean of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then mean values in period j are given by:

$$TX_{ij} = \frac{\sum_{i=1}^{I} TX_{ij}}{I}$$

`xclim.indices._simple.tx_min(`*tasmax: DataArray*, *freq: str = 'YS'*`)` → DataArray

Lowest max temperature.

The minimum of daily maximum temperature.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [same units as tasmax] – Minimum of daily maximum temperature.

Notes

Let TX_{ij} be the maximum temperature at day i of period j. Then the minimum daily maximum temperature for period j is:

$$TXn_j = min(TX_{ij})$$

xclim.indices._synoptic module

`xclim.indices._synoptic.jetstream_metric_woollings(`*ua: xarray.DataArray*`)`

Strength and latitude of jetstream.

Identify latitude and strength of maximum smoothed zonal wind speed in the region from 15 to 75°N and -60 to 0°E, using the formula outlined in ([Woollings2010]).

Warning: This metric expects eastward wind component (u) to be on a regular grid (i.e. Plate Carree, 1D lat and lon)

Parameters

ua (*xarray.DataArray*) – Eastward wind component (u) at between 750 and 950 hPa.

Returns

(xarray.DataArray, xarray.DataArray) – Daily time series of latitude of jetstream and Daily time series of strength of jetstream.

References

xclim.indices._threshold module

`xclim.indices._threshold.calm_days(`*sfcWind: DataArray, thresh: str = '2 m s-1', freq: str = 'MS'*`)` → DataArray

Calm days.

The number of days with average near-surface wind speed below threshold.

Parameters

- **sfcWind** (*xarray.DataArray*) – Daily windspeed.
- **thresh** (*str*) – Threshold average near-surface wind speed on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Number of days with average near-surface wind speed below threshold.

Notes

Let WS_{ij} be the windspeed at day i of period j. Then counted is the number of days where:

$$WS_{ij} < Threshold[ms-1]$$

`xclim.indices._threshold.cold_spell_days(`*tas: DataArray, thresh: str = '-10 degC', window: int = 5, freq: str = 'AS-JUL'*`)` → DataArray

Cold spell days.

The number of days that are part of cold spell events, defined as a sequence of consecutive days with mean daily temperature below a threshold in °C.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature below which a cold spell begins.
- **window** (*int*) – Minimum number of days with temperature below threshold to qualify as a cold spell.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Cold spell days.

Notes

Let T_i be the mean daily temperature on day i, the number of cold spell days during period ϕ is given by

$$\sum_{i\in\phi} \prod_{j=i}^{i+5} [T_j < thresh]$$

where $[P]$ is 1 if P is true, and 0 if false.

`xclim.indices._threshold.cold_spell_frequency(`*tas: DataArray*, *thresh: str = '-10 degC'*, *window: int = 5*, *freq: str = 'AS-JUL'*`)` → DataArray

Cold spell frequency.

The number of cold spell events, defined as a sequence of consecutive days with mean daily temperature below a threshold.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature below which a cold spell begins.
- **window** (*int*) – Minimum number of days with temperature below threshold to qualify as a cold spell.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Cold spell frequency.

`xclim.indices._threshold.continuous_snow_cover_end(`*snd: DataArray*, *thresh: str = '2 cm'*, *window: int = 14*, *freq: str = 'AS-JUL'*`)` → DataArray

End date of continuous snow cover.

First day after the start of the continuous snow cover when snow depth is below *threshold* for at least *window* consecutive days. WARNING: The default *freq* is valid for the northern hemisphere.

Parameters

- **snd** (*xarray.DataArray*) – Surface snow thickness.
- **thresh** (*str*) – Threshold snow thickness.
- **window** (*int*) – Minimum number of days with snow depth below threshold.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – First day after the start of the continuous snow cover when the snow depth goes below a threshold for a minimum duration. If there is no such day, return np.nan.

References

Chaumont D., Mailhot A., Diaconescu E.P., Fournier É., Logan T. 2017: Élaboration du portrait bioclimatique futur du Nunavik – Tome II. [Rapport présenté au Ministère de la forêt, de la faune et des parcs], Ouranos.

`xclim.indices._threshold.continuous_snow_cover_start`(*snd: DataArray*, *thresh: str = '2 cm'*, *window: int = 14*, *freq: str = 'AS-JUL'*) → DataArray

Start date of continuous snow cover.

Day of year when snow depth is above or equal *threshold* for at least *window* consecutive days. WARNING: The default *freq* is valid for the northern hemisphere.

Parameters

- **snd** (*xarray.DataArray*) – Surface snow thickness.
- **thresh** (*str*) – Threshold snow thickness.
- **window** (*int*) – Minimum number of days with snow depth above or equal to threshold.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – First day of the year when the snow depth is superior to a threshold for a minimum duration. If there is no such day, return np.nan.

References

Chaumont D., Mailhot A., Diaconescu E.P., Fournier É., Logan T. 2017: Élaboration du portrait bioclimatique futur du Nunavik – Tome II. [Rapport présenté au Ministère de la forêt, de la faune et des parcs], Ouranos.

`xclim.indices._threshold.cooling_degree_days`(*tas: DataArray*, *thresh: str = '18 degC'*, *freq: str = 'YS'*) → DataArray

Cooling degree days.

Sum of degree days above the temperature threshold at which spaces are cooled.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Temperature threshold above which air is cooled.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time][temperature] – Cooling degree days

Notes

Let x_i be the daily mean temperature at day i. Then the cooling degree days above temperature threshold *thresh* over period ϕ is given by:

$$\sum_{i \in \phi} (x_i - thresh[x_i > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false.

`xclim.indices._threshold.daily_pr_intensity`(*pr: DataArray*, *thresh: str = '1 mm/day'*, *freq: str = 'YS'*) → DataArray

Average daily precipitation intensity.

Return the average precipitation over wet days.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **thresh** (*str*) – Precipitation value over which a day is considered wet.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [precipitation] – The average precipitation over wet days for each period

Notes

Let $\mathbf{p} = p_0, p_1, \ldots, p_n$ be the daily precipitation and *thresh* be the precipitation threshold defining wet days. Then the daily precipitation intensity is defined as

$$\frac{\sum_{i=0}^{n} p_i [p_i \leq thresh]}{\sum_{i=0}^{n} [p_i \leq thresh]}$$

where $[P]$ is 1 if P is true, and 0 if false.

Examples

The following would compute for each grid cell of file *pr.day.nc* the average precipitation fallen over days with precipitation >= 5 mm at seasonal frequency, ie DJF, MAM, JJA, SON, DJF, etc.:

```
>>> from xclim.indices import daily_pr_intensity
>>> pr = xr.open_dataset(path_to_pr_file).pr
>>> daily_int = daily_pr_intensity(pr, thresh="5 mm/day", freq="QS-DEC")
```

`xclim.indices._threshold.days_with_snow`(*prsn: DataArray*, *low: str = '0 kg m-2 s-1'*, *high: str = '1E6 kg m-2 s-1'*, *freq: str = 'AS-JUL'*) → DataArray

Days with snow.

Return the number of days where snowfall is within low and high thresholds.

Parameters

- **prsn** (*xr.DataArray*) – Solid precipitation flux.
- **low** (*float*) – Minimum threshold solid precipitation flux.

- **high** (*float*) – Maximum threshold solid precipitation flux.
- **freq** (*str*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling.

Returns
xarray.DataArray, [time] – Number of days where snowfall is between low and high thresholds.

References

Matthews, L., Andrey, J., & Picketts, I. (2017). Planning for Winter Road Maintenance in the Context of Climate Change, Weather, Climate, and Society, 9(3), 521-532, https://doi.org/10.1175/WCAS-D-16-0103.1

`xclim.indices._threshold.degree_days_exceedance_date`(*tas: DataArray*, *thresh: str = '0 degC'*, *sum_thresh: str = '25 K days'*, *op: str = '>'*, *after_date: Optional[DayOfYearStr] = None*, *freq: str = 'YS'*) → DataArray

Degree days exceedance date.

Day of year when the sum of degree days exceeds a threshold. Degree days are computed above or below a given temperature threshold.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base degree days evaluation.
- **sum_thresh** (*str*) – Threshold of the degree days sum.
- **op** (*{">", "gt", "<", "lt", ">=", "ge", "<=", "le"}*) – If equivalent to '>', degree days are computed as *tas - thresh* and if equivalent to '<', they are computed as *thresh - tas*.
- **after_date** (*str, optional*) – Date at which to start the cumulative sum. In "mm-dd" format, defaults to the start of the sampling period.
- **freq** (*str*) – Resampling frequency. If *after_date* is given, *freq* should be annual.

Returns
xarray.DataArray, [dimensionless] – Degree-days exceedance date.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j, T is the reference threshold and ST is the sum threshold. Then, starting at day :math:i_0:, the degree days exceedance date is the first day k such that

$$\begin{cases}ST < \sum_{i=i_0}^{k} \max(TG_{ij} - T, 0) & \text{if } op \text{ is '>'} \\ ST < \sum_{i=i_0}^{k} \max(T - TG_{ij}, 0) & \text{if } op \text{ is '<'}\end{cases}$$

The resulting k is expressed as a day of year.

Cumulated degree days have numerous applications including plant and insect phenology. See https://en.wikipedia.org/wiki/Growing_degree-day for examples.

`xclim.indices._threshold.dry_days`(*pr: DataArray*, *thresh: str = '0.2 mm/d'*, *freq: str = 'YS'*) → DataArray

Dry days.

The number of days with daily precipitation below threshold.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Number of days with daily precipitation below threshold.

Notes

Let PR_{ij} be the daily precipitation at day i of period j. Then counted is the number of days where:

$$\sum PR_{ij} < Threshold[mm/day]$$

`xclim.indices._threshold.first_day_above`(*tasmin: DataArray*, *thresh: str = '0 degC'*, *after_date: DayOfYearStr = '01-01'*, *window: int = 1*, *freq: str = 'YS'*) → DataArray

First day of temperatures superior to a threshold temperature.

Returns first day of period where a temperature is superior to a threshold over a given number of days, limited to a starting calendar date.

WARNING: The default date and freq are valid for the northern hemisphere.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **after_date** (*str*) – Date of the year after which to look for the first event. Should have the format '%m-%d'.
- **window** (*int*) – Minimum number of days with temperature above threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when minimum temperature is superior to a threshold over a given number of days for the first time. If there is no such day, returns np.nan.

`xclim.indices._threshold.first_day_below`(*tasmin: DataArray*, *thresh: str = '0 degC'*, *after_date: DayOfYearStr = '07-01'*, *window: int = 1*, *freq: str = 'YS'*) → DataArray

First day of temperatures inferior to a threshold temperature.

Returns first day of period where a temperature is inferior to a threshold over a given number of days, limited to a starting calendar date.

WARNING: The default date and freq are valid for the northern hemisphere.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **after_date** (*str*) – Date of the year after which to look for the first frost event. Should have the format '%m-%d'.
- **window** (*int*) – Minimum number of days with temperature below threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when minimum temperature is inferior to a threshold over a given number of days for the first time. If there is no such day, returns np.nan.

`xclim.indices._threshold.first_snowfall`(*prsn: DataArray*, *thresh: str = '0.5 mm/day'*, *freq: str = 'AS-JUL'*) → DataArray

First day with solid precipitation above a threshold.

Returns the first day of a period where the solid precipitation exceeds a threshold. WARNING: The default *freq* is valid for the northern hemisphere.

Parameters

- **prsn** (*xarray.DataArray*) – Solid precipitation flux.
- **thresh** (*str*) – Threshold precipitation flux on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – First day of the year when the solid precipitation is superior to a threshold. If there is no such day, returns np.nan.

References

Climate Projections for the National Capital Region (2020), Volume 1: Results and Interpretation for Key Climate Indices, Report 193600.00, Prepared for Ottawa by CBCL.

`xclim.indices._threshold.freshet_start`(*tas: DataArray*, *thresh: str = '0 degC'*, *window: int = 5*, *freq: str = 'YS'*) → DataArray

First day consistently exceeding threshold temperature.

Returns first day of period where a temperature threshold is exceeded over a given number of days.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **window** (*int*) – Minimum number of days with temperature above threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when temperature exceeds threshold over a given number of days for the first time. If there is no such day, return np.nan.

Notes

Let x_i be the daily mean temperature at day of the year i for values of i going from 1 to 365 or 366. The start date of the freshet is given by the smallest index i for which

$$\prod_{j=i}^{i+w}[x_j > thresh]$$

is true, where w is the number of days the temperature threshold should be exceeded, and $[P]$ is 1 if P is true, and 0 if false.

`xclim.indices._threshold.frost_free_season_end(`*tasmin: DataArray*, *thresh: str = '0.0 degC'*, *mid_date: DayOfYearStr = '07-01'*, *window: int = 5*, *freq: str = 'YS'*`)` → DataArray

End of the frost free season.

Day of the year of the start of a sequence of days with minimum temperatures consistently below a threshold, after a period with minimum temperatures consistently above the same threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **mid_date** (*str*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'.
- **window** (*int*) – Minimum number of days with temperature below threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when minimum temperature is inferior to a threshold over a given number of days for the first time. If there is no such day or if a frost free season is not detected, returns np.nan. If the frost free season does not end within the time period, returns the last day of the period.

`xclim.indices._threshold.frost_free_season_length(`*tasmin: xarray.DataArray*, *window: int = 5*, *mid_date: DayOfYearStr | None = '07-01'*, *thresh: str = '0.0 degC'*, *freq: str = 'YS'*`)` → xarray.DataArray

Frost free season length.

The number of days between the first occurrence of at least N (def: 5) consecutive days with minimum daily temperature above a threshold (default: 0℃) and the first occurrence of at least N (def 5) consecutive days with minimum daily temperature below the same threshold A mid date can be given to limit the earliest day the end of season can take. WARNING: The default freq and mid_date values are valid for the northern hemisphere.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **window** (*int*) – Minimum number of days with temperature above threshold to mark the beginning and end of frost free season.
- **mid_date** (*str, optional*) – Date the must be included in the season. It is the earliest the end of the season can be. If None, there is no limit.

- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Frost free season length.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then counted is the number of days between the first occurrence of at least N consecutive days with:

$$TN_{ij} >= 0$$

and the first subsequent occurrence of at least N consecutive days with:

$$TN_{ij} < 0$$

Examples

```
>>> from xclim.indices import frost_season_length
>>> tasmin = xr.open_dataset(path_to_tasmin_file).tasmin
```

# For the Northern Hemisphere: >>> ffsl_nh = frost_free_season_length(tasmin, freq="YS")

# If working in the Southern Hemisphere, one can use: >>> ffsl_sh = frost_free_season_length(tasmin, freq="AS-JUL")

`xclim.indices._threshold.frost_free_season_start`(*tasmin: DataArray, thresh: str = '0.0 degC', window: int = 5, freq: str = 'YS'*) → DataArray

Start of the frost free season.

Day of the year of the start of a sequence of days with minimum temperatures consistently above or equal to a threshold, after a period with minimum temperatures consistently above the same threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **window** (*int*) – Minimum number of days with temperature above threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [dimensionless] – Day of the year when minimum temperature is superior to a threshold over a given number of days for the first time. If there is no such day or if a frost free season is not detected, returns np.nan.

Notes

Let x_i be the daily mean temperature at day of the year i for values of i going from 1 to 365 or 366. The start date of the start of growing season is given by the smallest index i for which:

$$\prod_{j=i}^{i+w} [x_j >= thresh]$$

is true, where w is the number of days the temperature threshold should be met or exceeded, and $[P]$ is 1 if P is true, and 0 if false.

`xclim.indices._threshold.frost_season_length`(*tasmin: xarray.DataArray*, *window: int = 5*, *mid_date: DayOfYearStr | None = '01-01'*, *thresh: str = '0.0 degC'*, *freq: str = 'AS-JUL'*) → xarray.DataArray

Frost season length.

The number of days between the first occurrence of at least N (def: 5) consecutive days with minimum daily temperature under a threshold (default: 0°C) and the first occurrence of at least N (def 5) consecutive days with minimum daily temperature above the same threshold A mid date can be given to limit the earliest day the end of season can take. WARNING: The default freq and mid_date values are valid for the northern hemisphere.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **window** (*int*) – Minimum number of days with temperature below threshold to mark the beginning and end of frost season.
- **mid_date** (*str, optional*) – Date the must be included in the season. It is the earliest the end of the season can be. If None, there is no limit.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Frost season length.

Notes

Let TN_{ij} be the minimum temperature at day i of period j. Then counted is the number of days between the first occurrence of at least N consecutive days with:

$$TN_{ij} > 0$$

and the first subsequent occurrence of at least N consecutive days with:

$$TN_{ij} < 0$$

Examples

```
>>> from xclim.indices import frost_season_length
>>> tasmin = xr.open_dataset(path_to_tasmin_file).tasmin
```

# For the Northern Hemisphere: >>> fsl_nh = frost_season_length(tasmin, freq="AS-JUL")

# If working in the Southern Hemisphere, one can use: >>> fsl_sh = frost_season_length(tasmin, freq="YS")

`xclim.indices._threshold.growing_degree_days(`*tas: DataArray, thresh: str = '4.0 degC', freq: str = 'YS'*`)` → DataArray

Growing degree-days over threshold temperature value.

The sum of degree-days over the threshold temperature.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time][temperature] – The sum of growing degree-days above a given threshold.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then the growing degree days are:

$$GD4_j = \sum_{i=1}^{I}(TG_{ij} - 4 | TG_{ij} > 4)$$

`xclim.indices._threshold.growing_season_end(`*tas: DataArray, thresh: str = '5.0 degC', mid_date: DayOfYearStr = '07-01', window: int = 5, freq: str = 'YS'*`)` → DataArray

End of the growing season.

Day of the year of the start of a sequence of days with mean temperatures consistently below a threshold, after a period with mean temperatures consistently above the same threshold.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **mid_date** (*str*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'.
- **window** (*int*) – Minimum number of days with temperature below threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [dimensionless] – Day of the year when temperature is inferior to a threshold over a given number of days for the first time. If there is no such day or if a growing season is not detected, returns np.nan. If the growing season does not end within the time period, returns the last day of the period.

`xclim.indices._threshold.growing_season_length`(*tas: DataArray, thresh: str = '5.0 degC', window: int = 6, mid_date: DayOfYearStr = '07-01', freq: str = 'YS'*) → DataArray

Growing season length.

The number of days between the first occurrence of at least six consecutive days with mean daily temperature over a threshold (default: 5℃) and the first occurrence of at least six consecutive days with mean daily temperature below the same threshold after a certain date. (Usually July 1st in the northern emisphere and January 1st in the southern hemisphere.)

WARNING: The default calendar values are only valid for the northern hemisphere.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **window** (*int*) – Minimum number of days with temperature above threshold to mark the beginning and end of growing season.
- **mid_date** (*str*) – Date of the year after which to look for the end of the season. Should have the format '%m-%d'.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Growing season length.

Notes

Let TG_{ij} be the mean temperature at day i of period j. Then counted is the number of days between the first occurrence of at least 6 consecutive days with:

$$TG_{ij} > 5$$

and the first occurrence after 1 July of at least 6 consecutive days with:

$$TG_{ij} < 5$$

Examples

```
>>> from xclim.indices import growing_season_length
>>> tas = xr.open_dataset(path_to_tas_file).tas
```

# For the Northern Hemisphere: >>> gsl_nh = growing_season_length(tas, mid_date="07-01", freq="AS")

# If working in the Southern Hemisphere, one can use: >>> gsl_sh = growing_season_length(tas, mid_date="01-01", freq="AS-JUL")

`xclim.indices._threshold.growing_season_start`(*tas: DataArray*, *thresh: str = '5.0 degC'*, *window: int = 5*, *freq: str = 'YS'*) → DataArray

Start of the growing season.

Day of the year of the start of a sequence of days with mean temperatures consistently above or equal to a threshold, after a period with mean temperatures consistently above the same threshold.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **window** (*int*) – Minimum number of days with temperature above threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when temperature is superior to a threshold over a given number of days for the first time. If there is no such day or if a growing season is not detected, returns np.nan.

Notes

Let x_i be the daily mean temperature at day of the year i for values of i going from 1 to 365 or 366. The start date of the start of growing season is given by the smallest index i for which:

$$\prod_{j=i}^{i+w}[x_j >= thresh]$$

is true, where w is the number of days the temperature threshold should be met or exceeded, and $[P]$ is 1 if P is true, and 0 if false.

`xclim.indices._threshold.heat_wave_index`(*tasmax: DataArray*, *thresh: str = '25.0 degC'*, *window: int = 5*, *freq: str = 'YS'*) → DataArray

Heat wave index.

Number of days that are part of a heatwave, defined as five or more consecutive days over 25℃.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to designate a heatwave.
- **window** (*int*) – Minimum number of days with temperature above threshold to qualify as a heatwave.
- **freq** (*str*) – Resampling frequency.

Returns

DataArray, [time] – Heat wave index.

`xclim.indices._threshold.heating_degree_days`(*tas: DataArray*, *thresh: str = '17.0 degC'*, *freq: str = 'YS'*) → DataArray

Heating degree days.

Sum of degree days below the temperature threshold at which spaces are heated.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time][temperature] – Heating degree days index.

Notes

This index intentionally differs from its ECA&D equivalent: HD17. In HD17, values below zero are not clipped before the sum. The present definition should provide a better representation of the energy demand for heating buildings to the given threshold.

Let TG_{ij} be the daily mean temperature at day i of period j. Then the heating degree days are:

$$HD17_j = \sum_{i=1}^{I} (17 - TG_{ij}) | TG_{ij} < 17)$$

`xclim.indices._threshold.hot_spell_frequency(`*tasmax: DataArray*, *thresh_tasmax: str = '30 degC'*, *window: int = 3*, *freq: str = 'YS'*`)` → DataArray

Hot spell frequency.

Number of hot spells over a given period. A hot spell is defined as an event where the maximum daily temperature exceeds a specific threshold over a minimum number of days.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmax** (*str*) – The maximum temperature threshold needed to trigger a heatwave event.
- **window** (*int*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [dimensionless] – Number of heatwave at the wanted frequency

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities (Casati et al., 2013).

In Robinson (2001), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

Casati, B., A. Yagouti, and D. Chaumont, 2013: Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. J. Appl. Meteor. Climatol., 52, 2669–2698, https://doi.org/10.1175/JAMC-D-12-0341.1

Robinson, P.J., 2001: On the Definition of a Heat Wave. J. Appl. Meteor., 40, 762–775, https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

`xclim.indices._threshold.hot_spell_max_length`(*tasmax: DataArray*, *thresh_tasmax: str = '30 degC'*, *window: int = 1*, *freq: str = 'YS'*) → DataArray

Longest hot spell.

Longest spell of high temperatures over a given period.

The longest series of consecutive days with tasmax 30 °C. Here, there is no minimum threshold for number of days in a row that must be reached or exceeded to count as a spell. A year with zero +30 °C days will return a longest spell value of zero.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh_tasmax** (*str*) – The maximum temperature threshold needed to trigger a heatwave event.
- **window** (*int*) – Minimum number of days with temperatures above thresholds to qualify as a heatwave.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Maximum length of continuous hot days at the wanted frequency.

Notes

The thresholds of 22° and 25°C for night temperatures and 30° and 35°C for day temperatures were selected by Health Canada professionals, following a temperature–mortality analysis. These absolute temperature thresholds characterize the occurrence of hot weather events that can result in adverse health outcomes for Canadian communities (Casati et al., 2013).

In Robinson (2001), the parameters would be *thresh_tasmin=27.22, thresh_tasmax=39.44, window=2* (81F, 103F).

References

Casati, B., A. Yagouti, and D. Chaumont, 2013: Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. J. Appl. Meteor. Climatol., 52, 2669–2698, https://doi.org/10.1175/JAMC-D-12-0341.1

Robinson, P.J., 2001: On the Definition of a Heat Wave. J. Appl. Meteor., 40, 762–775, https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

`xclim.indices._threshold.last_snowfall`(*prsn: DataArray*, *thresh: str = '0.5 mm/day'*, *freq: str = 'AS-JUL'*) → DataArray

Last day with solid precipitation above a threshold.

Returns the last day of a period where the solid precipitation exceeds a threshold. WARNING: The default freq is valid for the northern hemisphere.

Parameters

- **prsn** (*xarray.DataArray*) – Solid precipitation flux.
- **thresh** (*str*) – Threshold precipitation flux on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Last day of the year when the solid precipitation is superior to a threshold. If there is no such day, returns np.nan.

References

Climate Projections for the National Capital Region (2020), Volume 1: Results and Interpretation for Key Climate Indices, Report 193600.00, Prepared for Ottawa by CBCL.

`xclim.indices._threshold.last_spring_frost(`*tas: DataArray, thresh: str = '0 degC', before_date: DayOfYearStr = '07-01', window: int = 1, freq: str = 'YS'*`)` → DataArray

Last day of temperatures inferior to a threshold temperature.

Returns last day of period where a temperature is inferior to a threshold over a given number of days and limited to a final calendar date.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **before_date** (*str,*) – Date of the year before which to look for the final frost event. Should have the format '%m-%d'.
- **window** (*int*) – Minimum number of days with temperature below threshold needed for evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Day of the year when temperature is inferior to a threshold over a given number of days for the first time. If there is no such day, returns np.nan.

`xclim.indices._threshold.maximum_consecutive_dry_days(`*pr: DataArray, thresh: str = '1 mm/day', freq: str = 'YS'*`)` → DataArray

Maximum number of consecutive dry days.

Return the maximum number of consecutive days within the period where precipitation is below a certain threshold.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **thresh** (*str*) – Threshold precipitation on which to base evaluation.

- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – The maximum number of consecutive dry days (precipitation < threshold per period).

Notes

Let $\mathbf{p} = p_0, p_1, \ldots, p_n$ be a daily precipitation series and *thresh* the threshold under which a day is considered dry. Then let $\mathbf{s}$ be the sorted vector of indices i where $[p_i < thresh] \neq [p_{i+1} < thresh]$, that is, the days when the precipitation crosses the threshold. Then the maximum number of consecutive dry days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[p_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indices._threshold.maximum_consecutive_frost_days(`*tasmin: DataArray*, *thresh: str = '0.0 degC'*, *freq: str = 'AS-JUL'*`)` → DataArray

Maximum number of consecutive frost days (Tn < 0℃).

The maximum number of consecutive days within the period where the temperature is under a certain threshold (default: 0°C). WARNING: The default freq value is valid for the northern hemisphere.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – The maximum number of consecutive frost days (tasmin < threshold per period).

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily minimum temperature series and *thresh* the threshold below which a day is considered a frost day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i < thresh] \neq [t_{i+1} < thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive frost free days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indices._threshold.maximum_consecutive_frost_free_days(`*tasmin: DataArray*, *thresh: str = '0 degC'*, *freq: str = 'YS'*`)` → DataArray

Maximum number of consecutive frost free days (Tn >= 0℃).

Return the maximum number of consecutive days within the period where the minimum temperature is above or equal to a certain threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – The maximum number of consecutive frost free days (tasmin >= threshold per period).

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily minimum temperature series and $thresh$ the threshold above or equal to which a day is considered a frost free day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i <= thresh] \neq [t_{i+1} <= thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive frost free days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} >= thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indices._threshold.maximum_consecutive_tx_days(`*tasmax: DataArray*, *thresh: str = '25 degC'*, *freq: str = 'YS'*`)` → DataArray

Maximum number of consecutive days with tasmax above a threshold (summer days).

Return the maximum number of consecutive days within the period where the maximum temperature is above a certain threshold.

Parameters

- **tasmax** (*xarray.DataArray*) – Max daily temperature.
- **thresh** (*str*) – Threshold temperature.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – The maximum number of days with tasmax > thresh per periods (summer days).

Notes

Let $\mathbf{t} = t_0, t_1, \ldots, t_n$ be a daily maximum temperature series and $thresh$ the threshold above which a day is considered a summer day. Let $\mathbf{s}$ be the sorted vector of indices i where $[t_i < thresh] \neq [t_{i+1} < thresh]$, that is, the days when the temperature crosses the threshold. Then the maximum number of consecutive dry days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[t_{s_j} > thresh]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indices._threshold.maximum_consecutive_wet_days`(*pr: DataArray*, *thresh: str = '1 mm/day'*, *freq: str = 'YS'*) → DataArray

Consecutive wet days.

Returns the maximum number of consecutive wet days.

Parameters

- **pr** (*xarray.DataArray*) – Mean daily precipitation flux.
- **thresh** (*str*) – Threshold precipitation on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – The maximum number of consecutive wet days.

Notes

Let $\mathbf{x} = x_0, x_1, \ldots, x_n$ be a daily precipitation series and $\mathbf{s}$ be the sorted vector of indices i where $[p_i > thresh] \neq [p_{i+1} > thresh]$, that is, the days when the precipitation crosses the *wet day* threshold. Then the maximum number of consecutive wet days is given by

$$\max(\mathbf{d}) \quad \text{where} \quad d_j = (s_j - s_{j-1})[x_{s_j} > 0^\circ C]$$

where $[P]$ is 1 if P is true, and 0 if false. Note that this formula does not handle sequences at the start and end of the series, but the numerical algorithm does.

`xclim.indices._threshold.rprctot`(*pr: DataArray*, *prc: DataArray*, *thresh: str = '1.0 mm/day'*, *freq: str = 'YS'*) → DataArray

Proportion of accumulated precipitation arising from convective processes.

Return the proportion of total accumulated precipitation due to convection on days with total precipitation exceeding a specified threshold during the given period.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **prc** (*xarray.DataArray*) – Daily convective precipitation.
- **thresh** (*str*) – Precipitation value over which a day is considered wet.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – The proportion of the total precipitation accounted for by convective precipitation for each period.

`xclim.indices._threshold.sea_ice_area`(*siconc: DataArray*, *areacello: DataArray*, *thresh: str = '15 pct'*) → DataArray

Total sea ice area.

Sea ice area measures the total sea ice covered area where sea ice concentration is above a threshold, usually set to 15%.

Parameters

- **siconc** (*xarray.DataArray*) – Sea ice concentration (area fraction).
- **areacello** (*xarray.DataArray*) – Grid cell area (usually over the ocean).

- **thresh** (*str*) – Minimum sea ice concentration for a grid cell to contribute to the sea ice extent.

Returns
xarray.DataArray, [length]^2 – Sea ice area.

Notes

To compute sea ice area over a subregion, first mask or subset the input sea ice concentration data.

References

What is the difference between sea ice area and extent

`xclim.indices._threshold.sea_ice_extent`(*siconc: DataArray*, *areacello: DataArray*, *thresh: str = '15 pct'*) → DataArray

Total sea ice extent.

Sea ice extent measures the *ice-covered* area, where a region is considered ice-covered if its sea ice concentration is above a threshold usually set to 15%.

Parameters

- **siconc** (*xarray.DataArray*) – Sea ice concentration (area fraction).
- **areacello** (*xarray.DataArray*) – Grid cell area.
- **thresh** (*str*) – Minimum sea ice concentration for a grid cell to contribute to the sea ice extent.

Returns
xarray.DataArray, [length]^2 – Sea ice extent.

Notes

To compute sea ice area over a subregion, first mask or subset the input sea ice concentration data.

References

What is the difference between sea ice area and extent

`xclim.indices._threshold.snow_cover_duration`(*snd: DataArray*, *thresh: str = '2 cm'*, *freq: str = 'AS-JUL'*) → DataArray

Number of days with snow depth above a threshold.

Number of days where surface snow depth is greater or equal to given threshold. WARNING: The default *freq* is valid for the northern hemisphere.

Parameters

- **snd** (*xarray.DataArray*) – Surface snow thickness.
- **thresh** (*str*) – Threshold snow thickness.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days where snow depth is greater or equal to threshold.

`xclim.indices._threshold.tg_days_above(`*tas: DataArray*, *thresh: str = '10.0 degC'*, *freq: str = 'YS'*`)`

Number of days with tas above a threshold.

Number of days where daily mean temperature exceeds a threshold.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days where tas > threshold.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then counted is the number of days where:

$$TG_{ij} > Threshold[]$$

`xclim.indices._threshold.tg_days_below(`*tas: DataArray*, *thresh: str = '10.0 degC'*, *freq: str = 'YS'*`)`

Number of days with tas below a threshold.

Number of days where daily mean temperature is below a threshold.

Parameters

- **tas** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days where tas < threshold.

Notes

Let TG_{ij} be the daily mean temperature at day i of period j. Then counted is the number of days where:

$$TG_{ij} < Threshold[]$$

`xclim.indices._threshold.tn_days_above(`*tasmin: DataArray*, *thresh: str = '20.0 degC'*, *freq: str = 'YS'*`)`

Number of days with tasmin above a threshold (number of tropical nights).

Number of days where daily minimum temperature exceeds a threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.

- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days where tasmin $>$ threshold.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} > Threshold[]$$

`xclim.indices._threshold.tn_days_below(`*tasmin: DataArray, thresh: str = '-10.0 degC', freq: str = 'YS'*`)` → DataArray

Number of days with tasmin below a threshold.

Number of days where daily minimum temperature is below a threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days where tasmin $<$ threshold.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} < Threshold[]$$

`xclim.indices._threshold.tropical_nights(`*tasmin: DataArray, thresh: str = '20.0 degC', freq: str = 'YS'*`)` → DataArray

Tropical nights.

The number of days with minimum daily temperature above threshold.

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days with minimum daily temperature above threshold.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} > Threshold[]$$

Warning: The *tropical_nights* indice is being deprecated in favour of *tn_days_above* with *thresh="20 degC"* by default. The indicator reflects this change. This indice will be removed in a future version of xclim.

`xclim.indices._threshold.tx_days_above`(*tasmax: DataArray*, *thresh: str = '25.0 degC'*, *freq: str = 'YS'*) → DataArray

Number of days with tasmax above a threshold (number of summer days).

Number of days where daily maximum temperature exceeds a threshold.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Number of days where tasmax > threshold (number of summer days).

Notes

Let TX_{ij} be the daily maximum temperature at day i of period j. Then counted is the number of days where:

$$TX_{ij} > Threshold[]$$

`xclim.indices._threshold.tx_days_below`(*tasmax: DataArray*, *thresh: str = '25.0 degC'*, *freq: str = 'YS'*)

Number of days with tmax below a threshold.

Number of days where daily maximum temperature is below a threshold.

Parameters

- **tasmax** (*xarray.DataArray*) – Maximum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [time] – Number of days where tasmin < threshold.

Notes

Let TN_{ij} be the daily minimum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} < Threshold[]$$

`xclim.indices._threshold.warm_day_frequency(`*tasmax: DataArray*, *thresh: str = '30 degC'*, *freq: str = 'YS'*`)` → DataArray

Frequency of extreme warm days.

Return the number of days with tasmax > thresh per period

Parameters

- **tasmax** (*xarray.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days with tasmax > threshold per period.

Notes

Let TX_{ij} be the daily maximum temperature at day i of period j. Then counted is the number of days where:

$$TN_{ij} > Threshold[]$$

`xclim.indices._threshold.warm_night_frequency(`*tasmin: DataArray*, *thresh: str = '22 degC'*, *freq: str = 'YS'*`)` → DataArray

Frequency of extreme warm nights.

Return the number of days with tasmin > thresh per period

Parameters

- **tasmin** (*xarray.DataArray*) – Minimum daily temperature.
- **thresh** (*str*) – Threshold temperature on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days with tasmin > threshold per period.

`xclim.indices._threshold.wetdays(`*pr: DataArray*, *thresh: str = '1.0 mm/day'*, *freq: str = 'YS'*`)` → DataArray

Wet days.

Return the total number of days during period with precipitation over threshold.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **thresh** (*str*) – Precipitation value over which a day is considered wet.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – The number of wet days for each period [day].

Examples

The following would compute for each grid cell of file *pr.day.nc* the number days with precipitation over 5 mm at the seasonal frequency, ie DJF, MAM, JJA, SON, DJF, etc.:

```
>>> from xclim.indices import wetdays
>>> pr = xr.open_dataset(path_to_pr_file).pr
>>> wd = wetdays(pr, thresh="5 mm/day", freq="QS-DEC")
```

`xclim.indices._threshold.wetdays_prop(`*pr: DataArray*, *thresh: str = '1.0 mm/day'*, *freq: str = 'YS'*`)` → DataArray

Proportion of wet days.

Return the proportion of days during period with precipitation over threshold.

Parameters

- **pr** (*xarray.DataArray*) – Daily precipitation.
- **thresh** (*str*) – Precipitation value over which a day is considered wet.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – The proportion of wet days for each period [1].

Examples

The following would compute for each grid cell of file *pr.day.nc* the proportion of days with precipitation over 5 mm at the seasonal frequency, ie DJF, MAM, JJA, SON, DJF, etc.:

```
>>> from xclim.indices import wetdays_prop
>>> pr = xr.open_dataset(path_to_pr_file).pr
>>> wd = wetdays_prop(pr, thresh="5 mm/day", freq="QS-DEC")
```

`xclim.indices._threshold.windy_days(`*sfcWind: DataArray*, *thresh: str = '10.8 m s-1'*, *freq: str = 'MS'*`)` → DataArray

Windy days.

The number of days with average near-surface wind speed above threshold.

Parameters

- **sfcWind** (*xarray.DataArray*) – Daily average near-surface wind speed.
- **thresh** (*str*) – Threshold average near-surface wind speed on which to base evaluation.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray, [time] – Number of days with average near-surface wind speed above threshold.

Notes

Let WS_{ij} be the windspeed at day i of period j. Then counted is the number of days where:

$$WS_{ij} >= Threshold[ms-1]$$

`xclim.indices._threshold.winter_storm`(*snd: DataArray*, *thresh: str = '25 cm'*, *freq: str = 'AS-JUL'*) → DataArray

Days with snowfall over threshold.

Number of days with snowfall accumulation greater or equal to threshold.

Parameters

- **snd** (*xarray.DataArray*) – Surface snow depth.
- **thresh** (*str*) – Threshold on snowfall accumulation require to label an event a *winter storm*.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray – Number of days per period identified as winter storms.

Notes

Snowfall accumulation is estimated by the change in snow depth.

xclim.indices.fwi module

Fire Weather Indices Submodule

This submodule defines the `xclim.indices.fire_season()`, `xclim.indices.drought_code()` and `xclim.indices.fire_weather_indexes()` indices, which are used by the eponym indicators. Users should read this module's documentation and the one of *fire_weather_ufunc*.

First adapted from Matlab code *CalcFWITimeSeriesWithStartup.m* from GFWED made for using MERRA2 data, which was a translation of FWI.vba of the Canadian Fire Weather Index system. Then, updated and synchronized with the R code of the cffdrs package. When given the correct parameters, the current code has an error below 3% when compared with the [GFWED2015] data.

Parts of the code and of the documentation in this submodule are directly taken from [cffdrs] which was published with the GPLv2 license.

Fire season

Fire weather indexes are iteratively computed, each day's value depending on the previous day indexes. Additionally and optionally, the codes are "shut down" (set to NaN) in winter. There are a few ways of computing this shut down and the subsequent spring start-up. The *fire_season* function allows for full control of that, replicating the *fireSeason* method in the R package. It produces a mask to be given a *season_mask* in the indicators. However, the *fire_weather_ufunc* and the indicators also accept a *season_method* parameter so the fire season can be computed inside the iterator. Passing *season_method=None* switches to an "always on" mode replicating the *fwi* method of the R package.

The fire season determination is based on three consecutive daily maximum temperature thresholds ([Wotton&Flannigan1993]_, [Lawson&Armitage2008]_). A "GFWED" method is also implemented. There, the 12h LST temperature is used instead of the daily maximum. The current implementation is slightly different from the description in [GFWED2015], but it replicates the Matlab code when *temp_start_thresh* and *temp_end_thresh* are both set to 6 degC. In xclim, the number of consecutive days, the start and end temperature thresholds and the snow depth threshold can all be modified.

Overwintering

Additionnaly, overwintering of the drought code is also directly implemented in `fire_weather_ufunc()`. The last drought_code of the season is kept in "winter" (where the fire season mask is False) and the precipitation is accumulated until the start of the next season. The first drought code is computed as a function of these instead of using the default DCStart value. Parameters to `_overwintering_drought_code()` are listed below. The code for the overwintering is based on [MBHFJ2020].

Finally, a mechanism for dry spring starts is implemented. For now, it is slightly different from what the GFWED, uses, but seems to agree with the state of the science of the CFS. When activated, the drought code and Duff-moisture codes are started in spring with a value that is function of the number of days since the last significative precipitation event. The conventional start value increased by that number of days times a "dry start" factor. Parameters are controlled in the call of the indices and `fire_weather_ufunc()`. Overwintering of the drought code overrides this mechanism if both are activated. GFWED use a more complex approach with an added check on the previous day's snow cover for determining "dry" points. Moreover, there, the start values are only the multiplication of a factor to the number of dry days.

Examples

The current litterature seems to agree that climate-oriented series of the fire weather indexes should be computed using only the longest fire season of each year and activatting the overwintering of the drought code and the "dry start" for the duff-moisture code. The following example uses reasonable parameters when computing over all of Canada.

Note: Here the example snippets use the _indices_ defined in this very module, but we always recommend using the _indicators_ defined in the *xc.atmos* module.

```
>>> ds = open_dataset("ERA5/daily_surface_cancities_1990-1993.nc")
>>> ds = ds.assign(
...     hurs=xclim.atmos.relative_humidity_from_dewpoint(ds=ds),
...     tas=xclim.core.units.convert_units_to(ds.tas, "degC"),
...     pr=xclim.core.units.convert_units_to(ds.pr, "mm/d"),
...     sfcWind=xclim.atmos.wind_speed_from_vector(ds=ds)[0],
... )
>>> season_mask = fire_season(
...     tas=ds.tas,
...     method="WF93",
...     freq="YS",
...     # Parameters below are at their default values, but listed here for explicitness.
...     temp_start_thresh="12 degC",
...     temp_end_thresh="5 degC",
...     temp_condition_days=3,
... )
>>> out_fwi = fire_weather_indexes(
```

(continues on next page)

(continued from previous page)

```
...     tas=ds.tas,
...     pr=ds.pr,
...     hurs=ds.hurs,
...     sfcWind=ds.sfcWind,
...     lat=ds.lat,
...     season_mask=season_mask,
...     overwintering=True,
...     dry_start="CFS",
...     prec_thresh="1.5 mm/d",
...     dmc_dry_factor=1.2,
...     # Parameters below are at their default values, but listed here for explicitness.
...     carry_over_fraction=0.75,
...     wetting_efficiency_fraction=0.75,
...     dc_start=15,
...     dmc_start=6,
...     ffmc_start=85,
... )
```

Similarly, the next lines calculate the fire weather indexes, but according to the parameters and options used in NASA's GFWED datasets. Here, no need to split the fire season mask from the rest of the computation as _all_ seasons are used, even the very short shoulder seasons.

```
>>> ds = open_dataset("FWI/GFWED_sample_2017.nc")
>>> out_fwi = fire_weather_indexes(
...     tas=ds.tas,
...     pr=ds.prbc,
...     snd=ds.snow_depth,
...     hurs=ds.rh,
...     sfcWind=ds.sfcwind,
...     lat=ds.lat,
...     season_method="GFWED",
...     overwintering=False,
...     dry_start="GFWED",
...     temp_start_thresh="6 degC",
...     temp_end_thresh="6 degC",
...     # Parameters below are at their default values, but listed here for explicitness.
...     temp_condition_days=3,
...     snow_condition_days=3,
...     dc_start=15,
...     dmc_start=6,
...     ffmc_start=85,
...     dmc_dry_factor=2,
... )
```

References

Codes:

Updated source code for calculating fire danger indexes in the Canadian Forest Fire Weather Index System, Y. Wang, K.R. Anderson, and R.M. Suddaby, INFORMATION REPORT NOR-X-424, 2015.

https://cwfis.cfs.nrcan.gc.ca/background/dsm/fwi

Matlab code of the GFWED obtained through personal communication.

Fire season determination methods:

Drought Code overwintering:

`xclim.indices.fwi._convert_parameters(`*params: Mapping[str, int | float]*`)` → Mapping[str, int | float]

`xclim.indices.fwi._day_length(`*lat: int | float*, *mth: int*`)`

Return the average day length for a month within latitudinal bounds.

`xclim.indices.fwi._day_length_factor(`*lat: float*, *mth: int*`)`

Return the day length factor.

`xclim.indices.fwi._fire_season(`*tas: np.ndarray*, *snd: np.ndarray | None = None*, *method: str = 'WF93'*, *temp_start_thresh: float = 12*, *temp_end_thresh: float = 5*, *temp_condition_days: int = 3*, *snow_condition_days: int = 3*, *snow_thresh: float = 0.01*`)`

Compute the active fire season mask.

Parameters

- **tas** (*ndarray*) – Temperature [degC], the time axis on the last position.
- **snd** (*ndarray, optional*) – Snow depth [m], time axis on the last position, used with method == 'LA08'.
- **method** (*{"WF93", "LA08", "GFWED"}*) – Which method to use.
- **temp_start_thresh** (*float*) – Starting temperature threshold.
- **temp_end_thresh** (*float*) – Ending temperature threshold.
- **temp_condition_days** (*int*) – The number of days' temperature condition to consider.
- **snow_condition_days** (*int*) – The number of days' snow condition to consider.
- **snow_thresh** (*float*) – Numerical parameters of the methods.

Returns

ndarray [bool] – True where the fire season is active, same shape as tas.

`xclim.indices.fwi._fire_weather_calc(`*tas*, *pr*, *rh*, *ws*, *snd*, *mth*, *lat*, *season_mask*, *dc0*, *dmc0*, *ffmc0*, *winter_pr*, ***params*`)`

Primary function computing all Fire Weather Indexes. DO NOT CALL DIRECTLY, use *fire_weather_ufunc* instead.

`xclim.indices.fwi.build_up_index(`*dmc*, *dc*`)`

Build-up index.

Parameters

- **dmc** (*array*) – Duff moisture code.

- **dc** (*array*) – Drought code.

Returns
array – Build up index.

`xclim.indices.fwi.daily_severity_rating`(*fwi: np.ndarray*) → np.ndarry

Daily severity rating.

Parameters
fwi (*array_like*) – Fire weather index

Returns
array_like – Daily severity rating.

`xclim.indices.fwi.default_params = {'carry_over_fraction': 0.75, 'dc_dry_factor': 5, 'dc_start': 15, 'dmc_dry_factor': 2, 'dmc_start': 6, 'ffmc_start': 85, 'prec_thresh': (1.0, 'mm/d'), 'snow_condition_days': 3, 'snow_cover_days': 60, 'snow_min_cover_frac': 0.75, 'snow_min_mean_depth': (0.1, 'm'), 'snow_thresh': (0.01, 'm'), 'temp_condition_days': 3, 'temp_end_thresh': (5, 'degC'), 'temp_start_thresh': (12, 'degC'), 'wetting_efficiency_fraction': 0.75}`

Default values for numerical parameters of fire_weather_ufunc.

Parameters with units are given as a tuple of default value and units. A more complete explanation of these parameters is given in the doc of `fire_weather_ufunc()`.

`xclim.indices.fwi.drought_code`(*tas: xr.DataArray*, *pr: xr.DataArray*, *lat: xr.DataArray*, *snd: xr.DataArray | None = None*, *dc0: xr.DataArray | None = None*, *season_mask: xr.DataArray | None = None*, *season_method: str | None = None*, *overwintering: bool = False*, *dry_start: str | None = None*, *initial_start_up: bool = True*, ***params*)

Drought code (FWI component).

The drought code is part of the Canadian Forest Fire Weather Index System. It is a numeric rating of the average moisture content of organic layers.

Parameters

- **tas** (*xr.DataArray*) – Noon temperature.
- **pr** (*xr.DataArray*) – Rain fall in open over previous 24 hours, at noon.
- **lat** (*xr.DataArray*) – Latitude coordinate
- **snd** (*xr.DataArray*) – Noon snow depth.
- **dc0** (*xr.DataArray*) – Initial values of the drought code.
- **season_mask** (*xr.DataArray, optional*) – Boolean mask, True where/when the fire season is active.
- **season_method** (*{None, "WF93", "LA08", "GFWED"}*) – How to compute the start-up and shutdown of the fire season. If "None", no start-ups or shutdowns are computed, similar to the R fwi function. Ignored if *season_mask* is given.
- **overwintering** (*bool*) – Whether to activate DC overwintering or not. If True, either season_method or season_mask must be given.
- **dry_start** (*{None, "CFS", 'GFWED'}*) – Whether to activate the DC and DMC "dry start" mechanism and which method to use. , see `fire_weather_ufunc()`.

- **initial_start_up** (*bool*) – If True (default), grid points where the fire season is active on the first timestep go through a start_up phase for that time step. Otherwise, previous codes must be given as a continuing fire season is assumed for those points.
- **params** – Any other keyword parameters as defined in *xclim.indices.fwi.fire_weather_ufunc* and in `default_params`.

Returns
xr.DataArray, [dimensionless] – Drought code

Notes

See https://cwfis.cfs.nrcan.gc.ca/background/dsm/fwi, the module's doc and doc of `fire_weather_ufunc()` for more information.

References

Updated source code for calculating fire danger indexes in the Canadian Forest Fire Weather Index System, Y. Wang, K.R. Anderson, and R.M. Suddaby, INFORMATION REPORT NOR-X-424, 2015.

`xclim.indices.fwi.fire_season`(*tas: xr.DataArray, snd: xr.DataArray | None = None, method: str = 'WF93', freq: str | None = None, temp_start_thresh: str = '12 degC', temp_end_thresh: str = '5 degC', temp_condition_days: int = 3, snow_condition_days: int = 3, snow_thresh: str = '0.01 m'*)

Fire season mask.

Binary mask of the active fire season, defined by conditions on consecutive daily temperatures and, optionally, snow depths.

Parameters

- **tas** (*xr.DataArray*) – Daily surface temperature, cffdrs recommends using maximum daily temperature.
- **snd** (*xr.DataArray, optional*) – Snow depth, used with method == 'LA08'.
- **method** (*{"WF93", "LA08", "GFWED"}*) – Which method to use. "LA08" and "GFWED" need the snow depth.
- **freq** (*str, optional*) – If given only the longest fire season for each period defined by this frequency, Every "seasons" are returned if None, including the short shoulder seasons.
- **temp_start_thresh** (*str*) – Minimal temperature needed to start the season.
- **temp_end_thresh** (*str*) – Maximal temperature needed to end the season.
- **temp_condition_days** (*int*) – Number of days with temperature above or below the thresholds to trigger a start or an end of the fire season.
- **snow_condition_days** (*int*) – Parameters for the fire season determination. See `fire_season()`. Temperature is in degC, snow in m. The *snow_thresh* parameters is also used when *dry_start* is set to "GFWED".
- **snow_thresh** (*str*) – Minimal snow depth level to end a fire season, only used with method "LA08".

Returns
fire_season (*xr.DataArray*) – Fire season mask

References

[Wotton&Flannigan1993]_

[Lawson&Armitage2008]_

`xclim.indices.fwi.fire_weather_index(`*isi*, *bui*`)`

Fire weather index.

Parameters

- **isi** (*array*) – Initial spread index
- **bui** (*array*) – Build up index.

Returns

array – Build up index.

`xclim.indices.fwi.fire_weather_indexes(`*tas: xr.DataArray*, *pr: xr.DataArray*, *sfcWind: xr.DataArray*, *hurs: xr.DataArray*, *lat: xr.DataArray*, *snd: xr.DataArray | None = None*, *ffmc0: xr.DataArray | None = None*, *dmc0: xr.DataArray | None = None*, *dc0: xr.DataArray | None = None*, *season_mask: xr.DataArray | None = None*, *season_method: str | None = None*, *overwintering: bool = False*, *dry_start: str | None = None*, *initial_start_up: bool = True*, ***params*`)`

Fire weather indexes.

Computes the 6 fire weather indexes as defined by the Canadian Forest Service: the Drought Code, the Duff-Moisture Code, the Fine Fuel Moisture Code, the Initial Spread Index, the Build Up Index and the Fire Weather Index.

Parameters

- **tas** (*xr.DataArray*) – Noon temperature.
- **pr** (*xr.DataArray*) – Rain fall in open over previous 24 hours, at noon.
- **sfcWind** (*xr.DataArray*) – Noon wind speed.
- **hurs** (*xr.DataArray*) – Noon relative humidity.
- **lat** (*xr.DataArray*) – Latitude coordinate
- **snd** (*xr.DataArray*) – Noon snow depth, only used if *season_method='LA08'* is passed.
- **ffmc0** (*xr.DataArray*) – Initial values of the fine fuel moisture code.
- **dmc0** (*xr.DataArray*) – Initial values of the Duff moisture code.
- **dc0** (*xr.DataArray*) – Initial values of the drought code.
- **season_mask** (*xr.DataArray, optional*) – Boolean mask, True where/when the fire season is active.
- **season_method** (*{None, "WF93", "LA08", "GFWED"}*) – How to compute the start-up and shutdown of the fire season. If "None", no start-ups or shutdowns are computed, similar to the R fwi function. Ignored if *season_mask* is given.
- **overwintering** (*bool*) – Whether to activate DC overwintering or not. If True, either season_method or season_mask must be given.

- **dry_start** (*{None, 'CFS', 'GFWED'}*) – Whether to activate the DC and DMC "dry start" mechanism or not, see `fire_weather_ufunc()`.
- **initial_start_up** (*bool*) – If True (default), gridpoints where the fire season is active on the first timestep go through a start_up phase for that time step. Otherwise, previous codes must be given as a continuing fire season is assumed for those points.
- **params** – Any other keyword parameters as defined in `fire_weather_ufunc()` and in `default_params`.

Returns

- **DC** (*xr.DataArray, [dimensionless]*)
- **DMC** (*xr.DataArray, [dimensionless]*)
- **FFMC** (*xr.DataArray, [dimensionless]*)
- **ISI** (*xr.DataArray, [dimensionless]*)
- **BUI** (*xr.DataArray, [dimensionless]*)
- **FWI** (*xr.DataArray, [dimensionless]*)

Notes

See https://cwfis.cfs.nrcan.gc.ca/background/dsm/fwi, the module's doc and doc of `fire_weather_ufunc()` for more information.

References

Updated source code for calculating fire danger indexes in the Canadian Forest Fire Weather Index System, Y. Wang, K.R. Anderson, and R.M. Suddaby, INFORMATION REPORT NOR-X-424, 2015.

`xclim.indices.fwi.fire_weather_ufunc(`**, tas: xr.DataArray, pr: xr.DataArray, hurs: xr.DataArray | None = None, sfcWind: xr.DataArray | None = None, snd: xr.DataArray | None = None, lat: xr.DataArray | None = None, dc0: xr.DataArray | None = None, dmc0: xr.DataArray | None = None, ffmc0: xr.DataArray | None = None, winter_pr: xr.DataArray | None = None, season_mask: xr.DataArray | None = None, start_dates: str | xr.DataArray | None = None, indexes: Sequence[str] = None, season_method: str | None = None, overwintering: bool = False, dry_start: str | None = None, initial_start_up: bool = True, **params*`)`

Fire Weather Indexes computation using xarray's apply_ufunc.

No unit handling. Meant to be used by power users only. Please prefer using the `DC` and `FWI` indicators or the `drought_code()` and `fire_weather_indexes()` indices defined in the same submodule.

Dask arrays must have only one chunk along the "time" dimension. User can control which indexes are computed with the *indexes* argument.

Parameters

- **tas** (*xr.DataArray*) – Noon surface temperature in °C
- **pr** (*xr.DataArray*) – Rainfall over previous 24h, at noon in mm/day

- **hurs** (*xr.DataArray, optional*) – Noon surface relative humidity in %, not needed for DC
- **sfcWind** (*xr.DataArray, optional*) – Noon surface wind speed in km/h, not needed for DC, DMC or BUI
- **snd** (*xr.DataArray, optional*) – Noon snow depth in m, only needed if *season_method* is "LA08"
- **lat** (*xr.DataArray, optional*) – Latitude in °N, not needed for FFMC or ISI
- **dc0** (*xr.DataArray, optional*) – Previous DC map, see Notes. Defaults to NaN.
- **dmc0** (*xr.DataArray, optional*) – Previous DMC map, see Notes. Defaults to NaN.
- **ffmc0** (*xr.DataArray, optional*) – Previous FFMC map, see Notes. Defaults to NaN.
- **winter_pr** (*xr.DataArray, optional*) – Accumulated precipitation since the end of the last season, until the beginning of the current data, mm/day. Only used if *overwintering* is True, defaults to 0.
- **season_mask** (*xr.DataArray, optional*) – Boolean mask, True where/when the fire season is active.
- **indexes** (*Sequence[str], optional*) – Which indexes to compute. If intermediate indexes are needed, they will be added to the list and output.
- **season_method** (*{None, "WF93", "LA08", "GFWED"}*) – How to compute the start-up and shutdown of the fire season. If "None", no start-ups or shutdowns are computed, similar to the R fwi function. Ignored if *season_mask* is given.
- **overwintering** (*bool*) – Whether to activate DC overwintering or not. If True, either season_method or season_mask must be given.
- **dry_start** (*{None, 'CFS', 'GFWED'}*) – Whether to activate the DC and DMC "dry start" mechanism and which method to use. See Notes. If overwintering is activated, it overrides this parameter : only DMC is handled through the dry start mechanism.
- **initial_start_up** (*bool*) – If True (default), grid points where the fire season is active on the first timestep go through a start-up phase for that time step. Otherwise, previous codes must be given as a continuing fire season is assumed for those points.
- **carry_over_fraction** (*float*)
- **wetting_efficiency_fraction** (*float*) – Drought code overwintering parameters, see `overwintering_drought_code()`.
- **temp_start_thresh** (*float*) – Starting temperature threshold.
- **temp_end_thresh** (*float*) – Ending temperature threshold.
- **temp_condition_days** (*int*) – The number of days' temperature condition to consider.
- **snow_thresh** (*float*)
- **snow_condition_days** (*int*) – Parameters for the fire season determination. See `fire_season()`. Temperature is in degC, snow in m. The *snow_thresh* parameters is also used when *dry_start* is set to "GFWED", see Notes.
- **dc_start** (*float*)
- **dmc_start** (*float*)

- **ffmc_start** (*float*) – Default starting values for the three base codes.
- **prec_thresh** (*float*) – If the "dry start" is activated, this is the "wet" day precipitation threshold, see Notes. In mm/d.
- **dc_dry_factor** (*float*) – DC's start-up values for the "dry start" mechanism, see Notes.
- **dmc_dry_factor** (*float*) – DMC's start-up values for the "dry start" mechanism, see Notes.
- **snow_cover_days** (*int*)
- **snow_min_cover_frac** (*float*)
- **snow_min_mean_depth** (*float*) – Additional parameters for GFWED's version of the "dry start" mechanism. See Notes. Snow depth is in m.

Returns

dict[str, xarray.DataArray] – Dictionary containing the computed indexes as prescribed in *indexes*, including the intermediate ones needed, even if they were not explicitly listed in *indexes*. When overwintering is activated, *winter_pr* is added. If *season_method* is not None and *season_mask* was not given, *season_mask* is computed on-the-fly and added to the output.

Notes

When overwintering is activated, the argument *dc0* is understood as last season's last DC map and will be used to compute the overwintered DC at the beginning of the next season.

If overwintering is not activated and neither is fire season computation (*season_method* and *season_mask* are *None*), *dc0*, *dmc0* and *ffmc0* are understood as the codes on the day before the first day of FWI computation. They will default to their respective start values. This "always on" mode replicates the R "fwi" code.

If the "dry start" mechanism is set to "CFS" (but there is no overwintering), the arguments *dc0* and *dmc0* are understood as the potential start-up values from last season. With DC_{start} the conventional start-up value, F_{dry-dc} the *dc_dry_factor* and N_{dry} the number of days since the last significative precipitation event, the start-up value DC_0 is computed as:

$$DC_0 = DC_{start} + F_{dry-dc} * N_{dry}$$

The last significative precipitation event is the last day where precipitation was greater or equal to "prec_thresh". The same happens for the DMC, with corresponding parameters. If overwintering is activated, this mechanism is only used for the DMC.

Alternatively, *dry_start* can be set to "GFWED". In this mode, the start-up values are computed as:

$$DC_0 = F_{dry-dc} * N_{dry}$$

Where the current day is also included in the determination of N_{dry} (DC_0 can thus be 0). Finally, for this "GFWED" mode, if snow cover is provided, a second check is performed: the dry start procedure is skipped and conventional start-up values are used for cells where the snow cover of the last *snow_cover_days* was above *snow_thresh* for at least *snow_cover_days* * *snow_min_cover_frac* days and where the mean snow cover over the same period was greater of equal to *snow_min_mean_depth*.

`xclim.indices.fwi.initial_spread_index(`*ws: ndarray*, *ffmc: ndarray*`)` → ndarray

Initialize spread index.

Parameters

- **ws** (*array_like*) – Noon wind speed [km/h].
- **ffmc** (*array_like*) – Fine fuel moisture code.

Returns
array_like – Initial spread index.

`xclim.indices.fwi.overwintering_drought_code`(*last_dc: xr.DataArray*, *winter_pr: xr.DataArray*, *carry_over_fraction: xr.DataArray | float = 0.75*, *wetting_efficiency_fraction: xr.DataArray | float = 0.75*, *min_dc: xr.DataArray | float = 15*) → xr.DataArray

Compute the season-starting drought code based on the previous season's last drought code and the total winter precipitation.

This method replicates the "wDC" method of the [cffdrs] R package, with an added control on the "minimum" DC.

Parameters

- **last_dc** (*xr.DataArray*) – The previous season's last drought code.
- **winter_pr** (*xr.DataArray*) – The accumulated precipitation since the end of the fire season.
- **carry_over_fraction** (*xr.DataArray or float*) – Carry-over fraction of last fall's moisture
- **wetting_efficiency_fraction** (*xr.DataArray or float*) – Effectiveness of winter precipitation in recharging moisture reserves in spring
- **min_dc** (*xr.DataArray or float*) – Minimum drought code starting value.

Returns
wDC (*xr.DataArray*) – Overwintered drought code.

Notes

Details taken from the R package documentation ([cffdrs]): Of the three fuel moisture codes (i.e. FFMC, DMC and DC) making up the FWI System, only the DC needs to be considered in terms of its values carrying over from one fire season to the next. In Canada both the FFMC and the DMC are assumed to reach moisture saturation from overwinter precipitation at or before spring melt; this is a reasonable assumption and any error in these assumed starting conditions quickly disappears. If snowfall (or other overwinter precipitation) is not large enough however, the fuel layer tracked by the Drought Code may not fully reach saturation after spring snow melt; because of the long response time in this fuel layer (53 days in standard conditions) a large error in this spring starting condition can affect the DC for a significant portion of the fire season. In areas where overwinter precipitation is 200 mm or more, full moisture recharge occurs and DC overwintering is usually unnecessary. More discussion of overwintering and fuel drying time lag can be found in [Lawson&Armitage2008]_ and [VanWagner1985].

Carry-over fraction of last fall's moisture:

- 1.0, Daily DC calculated up to 1 November; continuous snow cover, or freeze-up, whichever comes first
- 0.75, Daily DC calculations stopped before any of the above conditions met or the area is subject to occasional winter chinook conditions, leaving the ground bare and subject to moisture depletion

- 0.5, Forested areas subject to long periods in fall or winter that favor depletion of soil moisture

Effectiveness of winter precipitation in recharging moisture reserves in spring:

- 0.9, Poorly drained, boggy sites with deep organic layers
- 0.75, Deep ground frost does not occur until late fall, if at all; moderately drained sites that allow infiltration of most of the melting snowpack
- 0.5, Chinook-prone areas and areas subject to early and deep ground frost; well-drained soils favoring rapid percolation or topography favoring rapid runoff before melting of ground frost

Source: [Lawson&Armitage2008]_ - Table 9.

References

[cffdrs]

[Lawson&Armitage2008]_

[VanWagner1985]

xclim.indices.generic module

Generic indices submodule

Helper functions for common generic actions done in the computation of indices.

`xclim.indices.generic.aggregate_between_dates(`*data: xr.DataArray, start: xr.DataArray | DayOfYearStr, end: xr.DataArray | DayOfYearStr, op: str = 'sum', freq: str | None = None*`)` → xr.DataArray

Aggregate the data over a period between start and end dates and apply the operator on the aggregated data.

Parameters

- **data** (*xr.DataArray*) – Data to aggregate between start and end dates.
- **start** (*xr.DataArray or DayOfYearStr*) – Start dates (as day-of-year) for the aggregation periods.
- **end** (*xr.DataArray or DayOfYearStr*) – End (as day-of-year) dates for the aggregation periods.
- **op** (*{'min', 'max', 'sum', 'mean', 'std'}*) – Operator.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray, [dimensionless] – Aggregated data between the start and end dates. If the end date is before the start date, returns np.nan. If there is no start and/or end date, returns np.nan.

`xclim.indices.generic.compare(`*da: xr.DataArray, op: str, thresh: float | int*`)` → xr.DataArray

Compare a dataArray to a threshold using given operator.

Parameters

- **da** (*xr.DataArray*) – Input data.

- **op** (*{">", "<", ">=", "<=", "gt", "lt", "ge", "le"}*) – Logical operator {>, <, >=, <=, gt, lt, ge, le }. e.g. arr > thresh.
- **thresh** (*Union[float, int]*) – Threshold value.

Returns
xr.DataArray – Boolean mask of the comparison.

`xclim.indices.generic.count_level_crossings(`*low_data: DataArray*, *high_data: DataArray*, *threshold: str*, *freq: str*`)` → DataArray

Calculate the number of times low_data is below threshold while high_data is above threshold.

First, the threshold is transformed to the same standard_name and units as the input data, then the thresholding is performed, and finally, the number of occurrences is counted.

Parameters

- **low_data** (*xr.DataArray*) – Variable that must be under the threshold.
- **high_data** (*xr.DataArray*) – Variable that must be above the threshold.
- **threshold** (*str*) – Quantity.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

`xclim.indices.generic.count_occurrences(`*data: DataArray*, *threshold: str*, *condition: str*, *freq: str*`)` → DataArray

Calculate the number of times some condition is met.

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, then this counts the number of times *data < threshold*. Finally, count the number of occurrences when condition is met.

Parameters

- **data** (*xr.DataArray*)
- **threshold** (*str*) – Quantity.
- **condition** (*{">", "<", ">=", "<=", "==", "!="}*) – Operator.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

`xclim.indices.generic.default_freq(`***indexer*`)` → str

Return the default frequency.

`xclim.indices.generic.degree_days(`*tas: DataArray*, *thresh: str*, *condition: str*`)` → DataArray

Calculate the degree days below/above the temperature threshold.

Parameters

- **tas** (*xr.DataArray*) – Mean daily temperature.
- **thresh** (*str*) – The temperature threshold.
- **condition** (*{"<", ">"}*) – Operator.

Returns
xarray.DataArray

`xclim.indices.generic.diurnal_temperature_range(`*low_ data: DataArray*, *high_ data: DataArray*, *reducer: str*, *freq: str*`)` → DataArray

Calculate the diurnal temperature range and reduce according to a statistic.

Parameters

- **low_data** (*xr.DataArray*) – The lowest daily temperature (tasmin).
- **high_data** (*xr.DataArray*) – The highest daily temperature (tasmax).
- **reducer** (*{'max', 'min', 'mean', 'sum'}*) – Reducer.
- **freq** (*str*) – Resampling frequency.

Returns

xarray.DataArray

`xclim.indices.generic.domain_count(`*da: DataArray*, *low: float*, *high: float*, *freq: str*`)` → DataArray

Count number of days where value is within low and high thresholds.

A value is counted if it is larger than *low*, and smaller or equal to *high*, i.e. in *]low, high]*.

Parameters

- **da** (*xr.DataArray*) – Input data.
- **low** (*float*) – Minimum threshold value.
- **high** (*float*) – Maximum threshold value.
- **freq** (*str*) – Resampling frequency defining the periods defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling.

Returns

xr.DataArray – The number of days where value is within [low, high] for each period.

`xclim.indices.generic.doymax(`*da: DataArray*`)` → DataArray

Return the day of year of the maximum value.

`xclim.indices.generic.doymin(`*da: DataArray*`)` → DataArray

Return the day of year of the minimum value.

`xclim.indices.generic.get_daily_events(`*da: DataArray*, *da_ value: float*, *operator: str*`)` → DataArray

Return a 0/1 mask when a condition is True or False.

Parameters

- **da** (*xr.DataArray*)
- **da_value** (*float*)
- **operator** (*{">", "<", ">=", "<=", "gt", "lt", "ge", "le"}*) – Logical operator {>, <, >=, <=, gt, lt, ge, le}. e.g. arr > thresh.

Notes

the function returns::

- 1 where operator(da, da_value) is True
- 0 where operator(da, da_value) is False
- nan where da is nan

Returns
xr.DataArray

`xclim.indices.generic.get_op(`*op: str*`)`

Get python's comparing function according to its name of representation.

Accepted op string are keys and values of xclim.indices.generic.binary_ops.

`xclim.indices.generic.interday_diurnal_temperature_range(`*low_data: DataArray*, *high_data: DataArray*, *freq: str*`)` → DataArray

Calculate the average absolute day-to-day difference in diurnal temperature range.

Parameters

- **low_data** (*xr.DataArray*) – The lowest daily temperature (tasmin).
- **high_data** (*xr.DataArray*) – The highest daily temperature (tasmax).
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

`xclim.indices.generic.last_occurrence(`*data: DataArray*, *threshold: str*, *condition: str*, *freq: str*`)` → DataArray

Calculate the last time some condition is met.

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Finally, locate the last occurrence when condition is met.

Parameters

- **data** (*xr.DataArray*)
- **threshold** (*str*) – Quantity
- **condition** (*{">", "<", ">=", "<=", "==", "!="}*) – Operator
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

`xclim.indices.generic.select_resample_op(`*da: DataArray*, *op: str*, *freq: str = 'YS'*, ***indexer*`)` → DataArray

Apply operation over each period that is part of the index selection.

Parameters

- **da** (*xr.DataArray*) – Input data.

- **op** (*str {'min', 'max', 'mean', 'std', 'var', 'count', 'sum', 'argmax', 'argmin'} or func*) – Reduce operation. Can either be a DataArray method or a function that can be applied to a DataArray.
- **freq** (*str*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling.
- **indexer** (*{dim: indexer, }, optional*) – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered.

Returns
xarray.DataArray – The maximum value for each period.

`xclim.indices.generic.statistics(`*data: DataArray*, *reducer: str*, *freq: str*`)` → DataArray

Calculate a simple statistic of the data.

Parameters

- **data** (*xr.DataArray*)
- **reducer** (*{'max', 'min', 'mean', 'sum'}*) – Reducer.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

`xclim.indices.generic.temperature_sum(`*data: DataArray*, *threshold: str*, *condition: str*, *freq: str*`)` → DataArray

Calculate the temperature sum above/below a threshold.

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Finally, the sum is calculated for those data values that fulfill the condition after subtraction of the threshold value. If the sum is for values below the threshold the result is multiplied by -1.

Parameters

- **data** (*xr.DataArray*)
- **threshold** (*str*) – Quantity
- **condition** (*{">", "<", ">=", "<=", "==", "!="}*) – Operator
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

`xclim.indices.generic.threshold_count(`*da: xr.DataArray*, *op: str*, *thresh: float | int | xr.DataArray*, *freq: str*`)` → xr.DataArray

Count number of days where value is above or below threshold.

Parameters

- **da** (*xr.DataArray*) – Input data.
- **op** (*{">", "<", ">=", "<=", "gt", "lt", "ge", "le"}*) – Logical operator {>, <, >=, <=, gt, lt, ge, le }. e.g. arr > thresh.
- **thresh** (*Union[float, int]*) – Threshold value.

- **freq** (*str*) – Resampling frequency defining the periods as defined in https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling.

Returns
xr.DataArray – The number of days meeting the constraints for each period.

`xclim.indices.generic.thresholded_statistics(`*data: DataArray*, *threshold: str*, *condition: str*, *reducer: str*, *freq: str*`)` → DataArray

Calculate a simple statistic of the data for which some condition is met.

First, the threshold is transformed to the same standard_name and units as the input data. Then the thresholding is performed as condition(data, threshold), i.e. if condition is <, data < threshold. Finally, the statistic is calculated for those data values that fulfill the condition.

Parameters

- **data** (*xr.DataArray*)
- **threshold** (*str*) – Quantity.
- **condition** (*{">", "<", ">=", "<=", "==", "!="}*) – Operator
- **reducer** (*{'max', 'min', 'mean', 'sum'}*) – Reducer.
- **freq** (*str*) – Resampling frequency.

Returns
xarray.DataArray

xclim.indices.helpers module

Helper functions submodule

Functions that encapsulate some geophysical logic but could be shared by many indices.

`xclim.indices.helpers.cosine_of_solar_zenith_angle(`*declination: DataArray*, *lat: DataArray*, *lon: Optional[DataArray] = None*, *time_correction: Optional[DataArray] = None*, *hours: Optional[DataArray] = None*, *interval: Optional[int] = None*, *stat: str = 'integral'*`)` → DataArray

Cosine of the solar zenith angle.

The solar zenith angle is the angle between a vertical line (perpendicular to the ground) and the sun rays. This function computes a daily statistic of its cosine : its integral from sunrise to sunset or the average over the same period. Based on [Kalogirou14]. In addition it computes instantaneuos values of tis cosine. Based on [Napoli20].

Parameters

- **declination** (*xr.DataArray*) – Solar declination. See `solar_declination()`.
- **lat** (*xr.DataArray*) – Latitude.
- **lon** (*xr.DataArray, optional*) – Longitude This is necessary if stat is "instant", "interval" or "sunlit".
- **time_correction** (*xr.DataArray, optional*) – Time correction for solar angle. See `time_correction_for_solar_angle()` This is necessary if stat is "instant".

- **hours** (*xr.DataArray, optional*) – Watch time hours. This is necessary if stat is “instant”, “interval” or “sunlit”.
- **interval** (*int, optional*) – Time interal between two time steps in hours This is necessary if stat is “interval” or “sunlit”.
- **stat** (*{‘integral’, ‘average’, ‘instant’, ‘interval’, ‘sunlit’}*) – Which daily statistic to return. If “integral”, this returns the integral of the cosine of the zenith angle from sunrise to sunset. If “average”, the integral is divided by the “duration” from sunrise to sunset. If “instant”, this returns the instantaneous cosine of the zenith angle. If “interval”, this returns the cosine of the zenith angle during each interval. If “sunlit”, this returns the cosine of the zenith angle during the sunlit period of each interval.

Returns

Cosine of the solar zenith angle, [rad] or [dimensionless] – If stat is “integral”, dimensions can be said to be “time” as the integral is on the hour angle. For seconds, multiply by the number of seconds in a comple day cycle (24*60*60) and divide by 2.

Notes

This code was inspired by the *thermofeel* and *PyWBGT* package.

References

Kalogirou, S. A. (2014). Chapter 2 — Environmental Characteristics. In S. A. Kalogirou (Ed.), Solar Energy Engineering (Second Edition) (pp. 51–123). Academic Press. https://doi.org/10.1016/B978-0-12-397270-5.00002-9 Di Napoli, C., Hogan, R.J. & Pappenberger, F. Mean radiant temperature from global-scale numerical weather prediction models. Int J Biometeorol 64, 1233–1245 (2020). https://doi.org/10.1007/s00484-020-01900-5

`xclim.indices.helpers.day_lengths`(*dates: DataArray*, *lat: DataArray*, *method: str = 'spencer'*) → DataArray

Day-lengths according to latitude and day of year.

See `solar_declination()` for the approximation used to compute the solar declination angle. Based on [Kalogirou14].

Parameters

- **dates** (*xr.DataArray*)
- **lat** (*xarray.DataArray*) – Latitude coordinate.
- **method** (*{‘spencer’, ‘simple’}*) – Which approximation to use when computing the solar declination angle. See `solar_declination()`.

Returns

xarray.DataArray, [hours] – Day-lengths in hours per individual day.

References

Kalogirou, S. A. (2014). Chapter 2 — Environmental Characteristics. In S. A. Kalogirou (Ed.), Solar Energy Engineering (Second Edition) (pp. 51–123). Academic Press. https://doi.org/10.1016/B978-0-12-397270-5.00002-9

`xclim.indices.helpers.distance_from_sun(`*dates: xr.DaraArray*`)` → xr.DataArray

Sun-earth distance.

The distance from sun to earth in astronomical units.

Parameters
dates (*xr.DataArray*) – Series of dates and time of days

Returns
xr.DataArray, [astronomical units] – Sun-earth distance

References

U.S. Naval Observatory:Astronomical Almanac. Washington, D.C.: U.S. Government Printing Office (1985).

`xclim.indices.helpers.eccentricity_correction_factor(`*day_angle: DataArray*, *method='spencer'*`)`

Eccentricity correction factor of the Earth's orbit.

The squared ratio of the mean distance Earth-Sun to the distance at a specific moment. As approximated by [Spencer1971].

Parameters

- **day_angle** (*xr.DataArray*) – Assuming the earth makes a full circle in a year, this is the angle covered from the beginning of the year up to that timestep. Also called the "julian day fraction". See `datetime_to_decimal_year()`.
- **method** – Which approximation to use. The default ("spencer") uses the first five terms of the fourier series of the eccentrencity, while "simple" approximates with only the first two.

Returns
Eccentricity correction factor, [dimensionless]

References

Spencer JW (1971) Fourier series representation of the position of the sun. Search 2(5):172

`xclim.indices.helpers.extraterrestrial_solar_radiation(`*times: DataArray*, *lat: DataArray*, *solar_constant: str = '1361 W m-2'*, *method='spencer'*`)` → DataArray

Extraterrestrial solar radiation.

This is the daily energy received on a surface parallel to the ground at the mean distance of the earth to the sun. It neglects the effect of the atmosphere. Computation is based on [Kalogirou14] and the default solar constant is taken from [Matthes17].

Parameters

- **times** (*xr.DataArray*) – Daily datetime data. This function makes no sense with data of other frequency.

- **lat** (*xr.DataArray*) – Latitude.
- **solar_constant** (*str*) – The solar constant, the energy received on earth from the sun per surface per time.
- **method** (*{'spencer', 'simple'}*) – Which method to use when computing the solar declination and the eccentricity correction factor. See `solar_declination()` and `eccentricity_correction_factor()`.

Returns
Extraterrestrial solar radiation, [J m-2 d-1]

References

`xclim.indices.helpers.solar_declination(`*day_angle: DataArray*, *method='spencer'*`)` → DataArray

Solar declination.

The angle between the sun rays and the earth's equator, in radians, as approximated by [Spencer1971] or assuming the orbit is a cirle.

Parameters

- **day_angle** (*xr.DataArray*) – Assuming the earth makes a full circle in a year, this is the angle covered from the beginning of the year up to that timestep. Also called the "julian day fraction". See `datetime_to_decimal_year()`.
- **method** (*{'spencer', 'simple'}*) – Which approximation to use. The default ("spencer") uses the first 7 terms of the Fourier series representing the observed declination, while "simple" assumes the orbit is a circle with a fixed obliquity and that the solstice/equinox happen at fixed angles on the orbit (the exact calendar date changes for leap years).

Returns
Solar declination angle, [rad]

References

`xclim.indices.helpers.time_correction_for_solar_angle(`*day_angle: DataArray*`)` → DataArray

Time correction for solar angle.

Every 1° of angular rotation on earth is equal to 4 minutes of time. The time correction helpsis needed to correct local watch time to solar time.

Parameters
day_angle (*xr.DataArray*) – Assuming the earth makes a full circle in a year, this is the angle covered from the beginning of the year up to that timestep. Also called the "julian day fraction". See `datetime_to_decimal_year()`.

Returns
Time correction of solar angle, [rad]

References

Di Napoli, C., Hogan, R.J. & Pappenberger, F. Mean radiant temperature from global-scale numerical weather prediction models. Int J Biometeorol 64, 1233–1245 (2020). https://doi.org/10.1007/s00484-020-01900-5

xclim.indices.run_length module

Run length algorithms submodule

Computation of statistics on runs of True values in boolean arrays.

`xclim.indices.run_length._rle_1d`(*ia*)

`xclim.indices.run_length.first_run`(*da: xr.DataArray*, *window: int*, *dim: str = 'time'*, *coord: str | bool | None = False*, *ufunc_1dim: str | bool = 'from_context'*) → xr.DataArray

Return the index of the first item of the first run of at least a given length.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values. When equal to 1, an optimized version of the algorithm is used.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **coord** (*Optional[str]*) – If not False, the function returns values along *dim* instead of indexes. If *dim* has a datetime dtype, *coord* can also be a str of the name of the DateTimeAccessor object to use (ex: 'dayofyear').
- **ufunc_1dim** (*Union[str, bool]*) – Use the 1d 'ufunc' version of this function : default (auto) will attempt to select optimal usage based on number of data points. Using 1D_ufunc=True is typically more efficient for DataArray with a small number of grid points. Ignored when *window=1*. It can be modified globally through the "run_length_ufunc" global option.

Returns

xr.DataArray – Index (or coordinate if *coord* is not False) of first item in first valid run. Returns np.nan if there are no valid runs.

`xclim.indices.run_length.first_run_1d`(*arr: Sequence[int | float]*, *window: int*) → int

Return the index of the first item of a run of at least a given length.

Parameters

- **arr** (*Sequence[Union[int, float]]*) – Input array.
- **window** (*int*) – Minimum duration of consecutive run to accumulate values.

Returns

int – Index of first item in first valid run. Returns np.nan if there are no valid runs.

`xclim.indices.run_length.first_run_after_date`(*da: xr.DataArray*, *window: int*, *date: DayOfYearStr | None = '07-01'*, *dim: str = 'time'*, *coord: bool | str | None = 'dayofyear'*) → xr.DataArray

Return the index of the first item of the first run after a given date.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values.
- **date** (*DayOfYearStr*) – The date after which to look for the run.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **coord** (*Optional[Union[bool, str]]*) – If not False, the function returns values along *dim* instead of indexes. If *dim* has a datetime dtype, *coord* can also be a str of the name of the DateTimeAccessor object to use (ex: 'dayofyear').

Returns

xr.DataArray – Index (or coordinate if *coord* is not False) of first item in the first valid run. Returns np.nan if there are no valid runs.

`xclim.indices.run_length.first_run_ufunc(`*x: xr.DataArray | Sequence[bool]*, *window: int*, *dim: str*`)` → xr.DataArray

Dask-parallel version of first_run_1d, ie: the first entry in array of consecutive true values.

Parameters

- **x** (*Union[xr.DataArray, Sequence[bool]]*) – Input array (bool).
- **window** (*int*) – Minimum run length.
- **dim** (*str*) – The dimension along which the runs are found.

Returns

xr.DataArray – A function operating along the time dimension of a dask-array.

`xclim.indices.run_length.index_of_date(`*time: xr.DataArray*, *date: DateStr | DayOfYearStr | None*, *max_idxs: int | None = None*, *default: int = 0*`)` → np.ndarray

Get the index of a date in a time array.

Parameters

- **time** (*xr.DataArray*) – An array of datetime values, any calendar.
- **date** (*DayOfYearStr or DateStr, optional*) – A string in the "yyyy-mm-dd" or "mm-dd" format. If None, returns default.
- **max_idxs** (*int, optional*) – Maximum number of returned indexes.
- **default** (*int*) – Index to return if date is None.

Raises

`ValueError` – If there are most instances of *date* in *time* than *max_idxs*.

Returns

numpy.ndarray – 1D array of integers, indexes of *date* in *time*.

`xclim.indices.run_length.keep_longest_run(`*da: DataArray*, *dim: str = 'time'*`)` → DataArray

Keep the longest run along a dimension.

Parameters

- **da** (*xr.DataArray*) – Boolean array.
- **dim** (*str*) – Dimension along which to check for the longest run.

Returns

xr.DataArray – Boolean array similar to da but with only one run, the (first) longest.

`xclim.indices.run_length.last_run(`*da: xr.DataArray*, *window: int*, *dim: str = 'time'*, *coord: str | bool | None = False*, *ufunc_1dim: str | bool = 'from_context'*`)` → xr.DataArray

Return the index of the last item of the last run of at least a given length.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values. When equal to 1, an optimized version of the algorithm is used.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **coord** (*Optional[str]*) – If not False, the function returns values along *dim* instead of indexes. If *dim* has a datetime dtype, *coord* can also be a str of the name of the DateTimeAccessor object to use (ex: 'dayofyear').
- **ufunc_1dim** (*Union[str, bool]*) – Use the 1d 'ufunc' version of this function : default (auto) will attempt to select optimal usage based on number of data points. Using *1D_ufunc=True* is typically more efficient for a DataArray with a small number of grid points. Ignored when *window=1*. It can be modified globally through the "run_length_ufunc" global option.

Returns

xr.DataArray – Index (or coordinate if *coord* is not False) of last item in last valid run. Returns np.nan if there are no valid runs.

`xclim.indices.run_length.last_run_before_date(`*da: xr.DataArray*, *window: int*, *date: DayOfYearStr = '07-01'*, *dim: str = 'time'*, *coord: bool | str | None = 'dayofyear'*`)` → xr.DataArray

Return the index of the last item of the last run before a given date.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values.
- **date** (*DayOfYearStr*) – The date before which to look for the last event.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **coord** (*Optional[Union[bool, str]]*) – If not False, the function returns values along *dim* instead of indexes. If *dim* has a datetime dtype, *coord* can also be a str of the name of the DateTimeAccessor object to use (ex: 'dayofyear').

Returns

xr.DataArray – Index (or coordinate if *coord* is not False) of last item in last valid run. Returns np.nan if there are no valid runs.

`xclim.indices.run_length.lazy_indexing(`*da: xr.DataArray*, *index: xr.DataArray*, *dim: str | None = None*`)` → xr.DataArray

Get values of *da* at indices *index* in a NaN-aware and lazy manner.

Two case

Parameters

- **da** (*xr.DataArray*) – Input array. If not 1D, *dim* must be given and must not appear in index.

- **index** (*xr.DataArray*) – N-d integer indices, if da is not 1D, all dimensions of index must be in da
- **dim** (*str, optional*) – Dimension along which to index, unused if *da* is 1D, should not be present in *index*.

Returns
xr.DataArray – Values of *da* at indices *index*.

`xclim.indices.run_length.longest_run(`*da: xr.DataArray, dim: str = 'time', ufunc_1dim: str | bool = 'from_context', index: str = 'first'*`)` → xr.DataArray

Return the length of the longest consecutive run of True values.

Parameters

- **da** (*xr.DataArray*) – N-dimensional array (boolean)
- **dim** (*str*) – Dimension along which to calculate consecutive run; Default: 'time'.
- **ufunc_1dim** (*Union[str, bool]*) – Use the 1d 'ufunc' version of this function : default (auto) will attempt to select optimal usage based on number of data points. Using 1D_ufunc=True is typically more efficient for DataArray with a small number of grid points. It can be modified globally through the "run_length_ufunc" global option.
- **index** (*{'first', 'last'}*) – If 'first', the run length is indexed with the first element in the run. If 'last', with the last element in the run.

Returns
xr.DataArray – Length of the longest run of True values along dimension (int).

`xclim.indices.run_length.npts_opt = 9000`

Arrays with less than this number of data points per slice will trigger the use of the ufunc version of run lengths algorithms.

`xclim.indices.run_length.rle(`*da: DataArray, dim: str = 'time', index: str = 'first'*`)` → DataArray

Generate basic run length function.

Parameters

- **da** (*xr.DataArray*) – Input array.
- **dim** (*str*) – Dimension name.
- **index** (*{'first', 'last'}*) – If 'first' (default), the run length is indexed with the first element in the run. If 'last', with the last element in the run.

Returns
xr.DataArray – Values are 0 where da is False (out of runs).

`xclim.indices.run_length.rle_1d(`*arr: int | float | bool | Sequence[int | float | bool]*`)` → tuple[np.array, np.array, np.array]

Return the length, starting position and value of consecutive identical values.

Parameters
arr (*Sequence[Union[int, float, bool]]*) – Array of values to be parsed.

Returns

- **values** (*np.array*) – The values taken by arr over each run.
- **run lengths** (*np.array*) – The length of each run.
- **start position** (*np.array*) – The starting index of each run.

Examples

```
>>> from xclim.indices.run_length import rle_1d
>>> a = [1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3]
>>> rle_1d(a)
(array([1, 2, 3]), array([2, 4, 6]), array([0, 2, 6]))
```

`xclim.indices.run_length.rle_statistics(`*da: xr.DataArray, reducer: str = 'max', window: int = 1, dim: str = 'time', ufunc_1dim: str | bool = 'from_context', index: str = 'first'*`)` → xr.DataArray

Return the length of consecutive run of True values, according to a reducing operator.

Parameters

- **da** (*xr.DataArray*) – N-dimensional array (boolean).
- **reducer** (*str*) – Name of the reducing function.
- **window** (*int*) – Minimal length of consecutive runs to be included in the statistics.
- **dim** (*str*) – Dimension along which to calculate consecutive run; Default: 'time'.
- **ufunc_1dim** (*Union[str, bool]*) – Use the 1d 'ufunc' version of this function : default (auto) will attempt to select optimal usage based on number of data points. Using 1D_ufunc=True is typically more efficient for DataArray with a small number of grid points. It can be modified globally through the "run_length_ufunc" global option.
- **index** (*{'first', 'last'}*) – If 'first' (default), the run length is indexed with the first element in the run. If 'last', with the last element in the run.

Returns

xr.DataArray – Length of runs of True values along dimension, according to the reducing function (float) If there are no runs (but the data is valid), returns 0.

`xclim.indices.run_length.run_bounds(`*mask: xr.DataArray, dim: str = 'time', coord: bool | str | None = True*`)`

Return the start and end dates of boolean runs along a dimension.

Parameters

- **mask** (*xr.DataArray*) – Boolean array.
- **dim** (*str*) – Dimension along which to look for runs.
- **coord** (*bool or str*) – If True, return values of the coordinate, if a string, returns values from *dim.dt.<coord>*. if False, return indexes.

Returns

xr.DataArray – With `dim` reduced to "events" and "bounds". The events dim is as long as needed, padded with NaN or NaT.

`xclim.indices.run_length.run_end_after_date(`*da: xr.DataArray, window: int, date: DayOfYearStr = '07-01', dim: str = 'time', coord: bool | str | None = 'dayofyear'*`)` → xr.DataArray

Return the index of the first item after the end of a run after a given date.

The run must begin before the date.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).

- **window** (*int*) – Minimum duration of consecutive run to accumulate values.
- **date** (*str*) – The date after which to look for the end of a run.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **coord** (*Optional[Union[bool, str]]*) – If not False, the function returns values along *dim* instead of indexes. If *dim* has a datetime dtype, *coord* can also be a str of the name of the DateTimeAccessor object to use (ex: 'dayofyear').

Returns
xr.DataArray – Index (or coordinate if *coord* is not False) of last item in last valid run. Returns np.nan if there are no valid runs.

`xclim.indices.run_length.season(`*da: xr.DataArray*, *window: int*, *date: DayOfYearStr | None = None*, *dim: str = 'time'*, *coord: str | bool | None = False*`)` → xr.Dataset

Return the bounds of a season (along dim).

A "season" is a run of True values that may include breaks under a given length (*window*). The start is computed as the first run of *window* True values, then end as the first subsequent run of *window* False values. If a date is passed, it must be included in the season.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive values to start and end the season.
- **date** (*DayOfYearStr, optional*) – The date (in MM-DD format) that a run must include to be considered valid.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **coord** (*Optional[str]*) – If not False, the function returns values along *dim* instead of indexes. If *dim* has a datetime dtype, *coord* can also be a str of the name of the DateTimeAccessor object to use (ex: 'dayofyear').

Returns
xr.Dataset – "dim" is reduced to "season_bnds" with 2 elements : season start and season end, both indices of da[dim].

Notes

The run can include holes of False or NaN values, so long as they do not exceed the window size.

If a date is given, the season start and end are forced to be on each side of this date. This means that even if the "real" season has been over for a long time, this is the date used in the length calculation. Example : Length of the "warm season", where T > 25°C, with date = 1st August. Let's say the temperature is over 25 for all june, but july and august have very cold temperatures. Instead of returning 30 days (june), the function will return 61 days (july + june).

`xclim.indices.run_length.season_length(`*da: xr.DataArray*, *window: int*, *date: DayOfYearStr | None = None*, *dim: str = 'time'*`)` → xr.DataArray

Return the length of the longest semi-consecutive run of True values (optionally including a given date).

A "season" is a run of True values that may include breaks under a given length (*window*). The start is computed as the first run of *window* True values, then end as the first subsequent run of *window* False values. If a date is passed, it must be included in the season.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum duration of consecutive values to start and end the season.
- **date** (*DayOfYearStr, optional*) – The date (in MM-DD format) that a run must include to be considered valid.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').

Returns
xr.DataArray – Length of the longest run of True values along a given dimension (inclusive of a given date) without breaks longer than a given length.

Notes

The run can include holes of False or NaN values, so long as they do not exceed the window size.

If a date is given, the season end is forced to be later or equal to this date. This means that even if the "real" season has been over for a long time, this is the date used in the length calculation. Example : Length of the "warm season", where T > 25°C, with date = 1st August. Let's say the temperature is over 25 for all june, but july and august have very cold temperatures. Instead of returning 30 days (june), the function will return 61 days (july + june).

`xclim.indices.run_length.statistics_run_1d(`*arr: Sequence[bool]*, *reducer: str*, *window: int = 1*`)` → int

Return statistics on lengths of run of identical values.

Parameters

- **arr** (*Sequence[bool]*) – Input array (bool)
- **reducer** (*{'mean', 'sum', 'min', 'max', 'std'}*) – Reducing function name.
- **window** (*int*) – Minimal length of runs to be included in the statistics

Returns
int – Statistics on length of runs.

`xclim.indices.run_length.statistics_run_ufunc(`*x: xr.DataArray | Sequence[bool]*, *reducer: str*, *window: int = 1*, *dim: str = 'time'*`)` → xr.DataArray

Dask-parallel version of statistics_run_1d, ie: the {reducer} number of consecutive true values in array.

Parameters

- **x** (*Sequence[bool]*) – Input array (bool)
- **reducer** (*{'min', 'max', 'mean', 'sum', 'std'}*) – Reducing function name.
- **window** (*int*) – Minimal length of runs.
- **dim** (*str*) – The dimension along which the runs are found.

Returns
xr.DataArray – A function operating along the time dimension of a dask-array.

`xclim.indices.run_length.suspicious_run(`*arr: xr.DataArray*, *dim: str = 'time'*, *window: int = 10*, *op: str = '>'*, *thresh: float | None = None*`)` → xr.DataArray

Return True where the array contains has runs of identical values, vectorized version.

In opposition to other run length functions, here the output has the same shape as the input.

Parameters

- **arr** (*xr.DataArray*) – Array of values to be parsed.
- **dim** (*str*) – Dimension along which to check for runs (default: "time").
- **window** (*int*) – Minimum run length
- **thresh** (*float, optional*) – Threshold above which values are checked for identical values.
- **op** (*{">", ">=", "==", "<", "<=", "eq", "gt", "lt", "gteq", "lteq"}*) – Operator for threshold comparison, defaults to ">".

Returns

xarray.DataArray

`xclim.indices.run_length.suspicious_run_1d(`*arr: np.ndarray*, *window: int = 10*, *op: str = '>'*, *thresh: float | None = None*`)` → np.ndarray

Return True where the array contains a run of identical values.

Parameters

- **arr** (*numpy.ndarray*) – Array of values to be parsed.
- **window** (*int*) – Minimum run length
- **op** (*{">", ">=", "==", "<", "<=", "eq", "gt", "lt", "gteq", "lteq"}, optional*) – Operator for threshold comparison. Defaults to ">".
- **thresh** (*float, optional*) – Threshold above which values are checked for identical values.

Returns

numpy.ndarray – Whether or not the data points are part of a run of identical values.

`xclim.indices.run_length.use_ufunc(`*ufunc_1dim: bool | str*, *da: xr.DataArray*, *dim: str = 'time'*, *index: str = 'first'*`)` → bool

Return whether the ufunc version of run length algorithms should be used with this DataArray or not.

If ufunc_1dim is 'from_context', the parameter is read from xclim's global (or context) options. If it is 'auto', this returns False for dask-backed array and for arrays with more than `npts_opt` points per slice along *dim*.

Parameters

- **ufunc_1dim** (*{'from_context', 'auto', True, False}*) – The method for handling the ufunc parameters.
- **da** (*xr.DataArray*) – Input array.
- **dim** (*str*) – The dimension along which to find runs.
- **index** (*{'first', 'last'}*) – If 'first' (default), the run length is indexed with the first element in the run. If 'last', with the last element in the run.

Returns

bool – If ufunc_1dim is "auto", returns True if the array is on dask or too large. Otherwise, returns ufunc_1dim.

`xclim.indices.run_length.windowed_run_count(`*da: xr.DataArray*, *window: int*, *dim: str = 'time'*, *ufunc_1dim: str | bool = 'from_context'*, *index: str = 'first'*`)` → xr.DataArray

Return the number of consecutive true values in array for runs at least as long as given duration.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).
- **window** (*int*) – Minimum run length. When equal to 1, an optimized version of the algorithm is used.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **ufunc_1dim** (*Union[str, bool]*) – Use the 1d 'ufunc' version of this function : default (auto) will attempt to select optimal usage based on number of data points. Using 1D_ufunc=True is typically more efficient for DataArray with a small number of grid points. Ignored when *window=1*. It can be modified globally through the "run_length_ufunc" global option.
- **index** (*{'first', 'last'}*) – If 'first', the run length is indexed with the first element in the run. If 'last', with the last element in the run.

Returns

xr.DataArray – Total number of *True* values part of a consecutive runs of at least *window* long.

`xclim.indices.run_length.windowed_run_count_1d(`*arr: Sequence[bool]*, *window: int*`)` → int

Return the number of consecutive true values in array for runs at least as long as given duration.

Parameters

- **arr** (*Sequence[bool]*) – Input array (bool).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values.

Returns

int – Total number of true values part of a consecutive run at least *window* long.

`xclim.indices.run_length.windowed_run_count_ufunc(`*x: xr.DataArray | Sequence[bool]*, *window: int*, *dim: str*`)` → xr.DataArray

Dask-parallel version of windowed_run_count_1d, ie: the number of consecutive true values in array for runs at least as long as given duration.

Parameters

- **x** (*Sequence[bool]*) – Input array (bool).
- **window** (*int*) – Minimum duration of consecutive run to accumulate values.
- **dim** (*str*) – Dimension along which to calculate windowed run.

Returns

xr.DataArray – A function operating along the time dimension of a dask-array.

`xclim.indices.run_length.windowed_run_events(`*da: xr.DataArray*, *window: int*, *dim: str = 'time'*, *ufunc_1dim: str | bool = 'from_context'*, *index: str = 'first'*`)` → xr.DataArray

Return the number of runs of a minimum length.

Parameters

- **da** (*xr.DataArray*) – Input N-dimensional DataArray (boolean).

- **window** (*int*) – Minimum run length. When equal to 1, an optimized version of the algorithm is used.
- **dim** (*str*) – Dimension along which to calculate consecutive run (default: 'time').
- **ufunc_1dim** (*Union[str, bool]*) – Use the 1d 'ufunc' version of this function : default (auto) will attempt to select optimal usage based on number of data points. Using 1D_ufunc=True is typically more efficient for DataArray with a small number of grid points. Ignored when *window=1*. It can be modified globally through the "run_length_ufunc" global option.
- **index** (*{'first', 'last'}*) – If 'first', the run length is indexed with the first element in the run. If 'last', with the last element in the run.

Returns
xr.DataArray – Number of distinct runs of a minimum length (int).

`xclim.indices.run_length.windowed_run_events_1d(`*arr: Sequence[bool]*, *window: int*`)` → DataArray

Return the number of runs of a minimum length.

Parameters

- **arr** (*Sequence[bool]*) – Input array (bool).
- **window** (*int*) – Minimum run length.

Returns
xr.DataArray – Number of distinct runs of a minimum length.

`xclim.indices.run_length.windowed_run_events_ufunc(`*x: xr.DataArray | Sequence[bool]*, *window: int*, *dim: str*`)` → xr.DataArray

Dask-parallel version of windowed_run_events_1d, ie: the number of runs at least as long as given duration.

Parameters

- **x** (*Sequence[bool]*) – Input array (bool).
- **window** (*int*) – Minimum run length.
- **dim** (*str*) – Dimension along which to calculate windowed run.

Returns
xr.DataArray – A function operating along the time dimension of a dask-array.

xclim.indices.stats module

Statistic-related functions. See the *frequency_analysis* notebook for examples.

`xclim.indices.stats._fit_start(`*x*, *dist*, ***fitkwargs*`)` → tuple[tuple, dict]

Return initial values for distribution parameters.

Providing the ML fit method initial values can help the optimizer find the global optimum.

Parameters

- **x** (*array-like*) – Input data.
- **dist** (*str*) – Name of the univariate distribution, such as *beta*, *expon*, *genextreme*, *gamma*, *gumbel_r*, *lognorm*, *norm* (see scipy.stats). Only *genextreme* and *weibull_exp* distributions are supported.
- **fitkwargs** – Kwargs passed to fit.

Returns
tuple, dict

References

Coles, S., 2001. An Introduction to Statistical Modeling of Extreme Values. Springer-Verlag, London, U.K., 208pp Cohen & Whittle. 1988. Parameter Estimation in Reliability and Life Span Models, p. 25 ff, Marcel Dekker.

`xclim.indices.stats.fa(`*da: xr.DataArray*, *t: int | Sequence*, *dist: str = 'norm'*, *mode: str = 'max'*`)` → xr.DataArray

Return the value corresponding to the given return period.

Parameters

- **da** (*xr.DataArray*) – Maximized/minimized input data with a *time* dimension.
- **t** (*Union[int, Sequence]*) – Return period. The period depends on the resolution of the input data. If the input array's resolution is yearly, then the return period is in years.
- **dist** (*str*) – Name of the univariate distribution, such as *beta*, *expon*, *genextreme*, *gamma*, *gumbel_r*, *lognorm*, *norm* (see scipy.stats).
- **mode** (*{'min', 'max}*) – Whether we are looking for a probability of exceedance (max) or a probability of non-exceedance (min).

Returns
xarray.DataArray – An array of values with a 1/t probability of exceedance (if mode=='max').

`xclim.indices.stats.fit(`*da: DataArray*, *dist: str = 'norm'*, *method: str = 'ML'*, *dim: str = 'time'*, ***fitkwargs*`)` → DataArray

Fit an array to a univariate distribution along the time dimension.

Parameters

- **da** (*xr.DataArray*) – Time series to be fitted along the time dimension.
- **dist** (*str*) – Name of the univariate distribution, such as beta, expon, genextreme, gamma, gumbel_r, lognorm, norm (see scipy.stats for full list). If the PWM method is used, only the following distributions are currently supported: 'expon', 'gamma', 'genextreme', 'genpareto', 'gumbel_r', 'pearson3', 'weibull_min'.
- **method** (*{"ML", "PWM"}*) – Fitting method, either maximum likelihood (ML) or probability weighted moments (PWM), also called L-Moments. The PWM method is usually more robust to outliers.
- **dim** (*str*) – The dimension upon which to perform the indexing (default: "time").
- **fitkwargs** – Other arguments passed directly to `_fitstart()` and to the distribution's *fit*.

Returns
xr.DataArray – An array of fitted distribution parameters.

Notes

Coordinates for which all values are NaNs will be dropped before fitting the distribution. If the array still contains NaNs, the distribution parameters will be returned as NaNs.

`xclim.indices.stats.frequency_analysis(`*da: xr.DataArray*, *mode: str*, *t: int | Sequence[int]*, *dist: str*, *window: int = 1*, *freq: str | None = None*, ***indexer*`)` → xr.DataArray

Return the value corresponding to a return period.

Parameters

- **da** (*xarray.DataArray*) – Input data.
- **mode** (*{'min', 'max'}*) – Whether we are looking for a probability of exceedance (high) or a probability of non-exceedance (low).
- **t** (*int or sequence*) – Return period. The period depends on the resolution of the input data. If the input array's resolution is yearly, then the return period is in years.
- **dist** (*str*) – Name of the univariate distribution, such as *beta*, *expon*, *genextreme*, *gamma*, *gumbel_r*, *lognorm*, *norm* (see scipy.stats).
- **window** (*int*) – Averaging window length (days).
- **freq** (*str*) – Resampling frequency. If None, the frequency is assumed to be 'YS' unless the indexer is season='DJF', in which case *freq* would be set to *AS-DEC*.
- **indexer** (*{dim: indexer, }, optional*) – Time attribute and values over which to subset the array. For example, use season='DJF' to select winter values, month=1 to select January, or month=[6,7,8] to select summer months. If not indexer is given, all values are considered.

Returns

xarray.DataArray – An array of values with a 1/t probability of exceedance or non-exceedance when mode is high or low respectively.

`xclim.indices.stats.get_dist(`*dist*`)`

Return a distribution object from *scipy.stats*.

`xclim.indices.stats.get_lm3_dist(`*dist*`)`

Return a distribution object from *lmoments3.distr*.

`xclim.indices.stats.parametric_cdf(`*p: xr.DataArray*, *v: float | Sequence*`)` → xr.DataArray

Return the cumulative distribution function corresponding to the given distribution parameters and value.

Parameters

- **p** (*xr.DataArray*) – Distribution parameters returned by the *fit* function. The array should have dimension *dparams* storing the distribution parameters, and attribute *scipy_dist*, storing the name of the distribution.
- **v** (*Union[float, Sequence]*) – Value to compute the CDF.

Returns

xarray.DataArray – An array of parametric CDF values estimated from the distribution parameters.

`xclim.indices.stats.parametric_quantile(`*p: xr.DataArray, q: int | Sequence*`)` → xr.DataArray

Return the value corresponding to the given distribution parameters and quantile.

Parameters

- **p** (*xr.DataArray*) – Distribution parameters returned by the *fit* function. The array should have dimension *dparams* storing the distribution parameters, and attribute *scipy_ dist*, storing the name of the distribution.
- **q** (*Union[float, Sequence]*) – Quantile to compute, which must be between *0* and *1*, inclusive.

Returns

xarray.DataArray – An array of parametric quantiles estimated from the distribution parameters.

Notes

When all quantiles are above 0.5, the *isf* method is used instead of *ppf* because accuracy is sometimes better.

xclim.sdba package

Statistical Downscaling and Bias Adjustment

The *xclim.sdba* submodule provides bias-adjustment methods and will eventually provide statistical downscaling algorithms. Almost all adjustment algorithms conform to the *train* - *adjust* scheme, formalized within *TrainAdjust* classes. Given a reference time series (ref), historical simulations (hist) and simulations to be adjusted (sim), any bias-adjustment method would be applied by first estimating the adjustment factors between the historical simulation and the observations series, and then applying these factors to *sim*, which could be a future simulation:

```
# Create the adjustment object by training it with reference and model data, plus␣
↪certains arguments
Adj = Adjustment.train(ref, hist, group="time.month")
# Get a scenario by applying the adjustment to a simulated timeseries.
scen = Adj.adjust(sim, interp="linear")
Adj.ds.af  # adjustment factors.
```

The *group* argument allows adjustment factors to be estimated independently for different periods: the full time series, months, seasons or day of the year. The *interp* argument then allows for interpolation between these adjustment factors to avoid discontinuities in the bias-adjusted series (only applicable for monthly grouping).

Warning: If grouping according to the day of the year is needed, the `xclim.core.calendar` submodule contains useful tools to manage the different calendars that the input data can have. By default, if 2 different calendars are passed, the adjustment factors will always be interpolated to the largest range of day of the years but this can lead to strange values and we recommend converting the data beforehand to a common calendar.

The same interpolation principle is also used for quantiles. Indeed, for methods extracting adjustment factors by quantile, interpolation is also done between quantiles. This can help reduce discontinuities in the adjusted time series, and possibly reduce the number of quantile bins used.

Modular Approach

This module adopts a modular approach instead of implementing published and named methods directly. A generic bias adjustment process is laid out as follows:

- preprocessing on `ref`, `hist` and `sim` (using methods in *xclim.sdba.processing* or *xclim.sdba.detrending*)
- creating and training the adjustment object `Adj = Adjustment.train(obs, sim, **kwargs)` (from *xclim.sdba.adjustment*)
- adjustment `scen = Adj.adjust(sim, **kwargs)`
- post-processing on `scen` (for example: re-trending)

The train-adjust approach allows to inspect the trained adjustment object. The training information is stored in the underlying *Adj.ds* dataset and always has a *af* variable with the adjustment factors. Its layout and the other available variables vary between the different algorithm, refer to *Adjustment methods*.

Parameters needed by the training and the adjustment are saved to the `Adj.ds` dataset as a *adj_params* attribute. Other parameters, those only needed by the adjustment are passed in the *adjust* call and written to the history attribute in the output scenario dataarray.

Grouping

For basic time period grouping (months, day of year, season), passing a string to the methods needing it is sufficient. Most methods acting on grouped data also accept a *window* int argument to pad the groups with data from adjacent ones. Units of *window* are the sampling frequency of the main grouping dimension (usually *time*). For more complex grouping, one can pass an instance of *xclim.sdba.base.Grouper* directly.

Notes for Developers

To be scalable and performant, the sdba module makes use of the special decorators :py:func`xclim.sdba.base.map_blocks` and *xclim.sdba.base.map_groups()*. However, they have the inconvenient that functions wrapped by them are unable to manage xarray attributes (including units) correctly and their signatures are sometime wrong and often unclear. For this reason, the module is often divided in two parts : the (decorated) compute functions in a "private" file (ex: `_adjustment.py`) and the user-facing functions or objects in corresponding public file (ex: `adjustment.py`). See the *sdba-advanced* notebook for more info on the reasons for this move.

Other restrictions : `map_blocks` will remove any "auxiliary" coordinates before calling the wrapped function and will add them back on exit.

Submodules

xclim.sdba._adjustment module

Adjustment Algorithms

This file defines the different steps, to be wrapped into the Adjustment objects.

`xclim.sdba._adjustment._extremes_train_1d`(*ref*, *hist*, *ref_params*, ***, *q_thresh*, *cluster_thresh*, *dist*, *N*)

Train for method ExtremeValues, only for 1D input along time.

`xclim.sdba._adjustment._fit_cluster_and_cdf(`*data*, *thresh*, *dist*, *cluster_thresh*`)`

Fit 1D cluster maximums and immediately compute CDF.

`xclim.sdba._adjustment._fit_on_cluster(`*data*, *thresh*, *dist*, *cluster_thresh*`)`

Extract clusters on 1D data and fit "dist" on the maximums.

`xclim.sdba._adjustment.npdf_transform(`*ds: Dataset*, ***kwargs*`)` → Dataset

N-pdf transform : Iterative univariate adjustment in random rotated spaces.

Parameters

- **ds** (*xr.Dataset*) –

  Dataset variables:
  ref : Reference multivariate timeseries hist : simulated timeseries on the reference period sim : Simulated timeseries on the projected period. rot_matrices : Random rotation matrices.

- **kwargs** – pts_dim : multivariate dimension name base : Adjustment class base_kws : Kwargs for initialising the adjustment object adj_kws : Kwargs of the *adjust* call n_escore : Number of elements to include in the e_score test (0 for all, < 0 to skip)

Returns

xr.Dataset – Dataset with *scenh*, *scens* and *escores* DataArrays, where *scenh* and *scens* are *hist* and *sim* respectively after adjustment according to *ref*. If *n_escore* is negative, *escores* will be filled with NaNs.

xclim.sdba._processing module

Compute functions of processing.py.

Here are defined the functions wrapped by map_blocks or map_groups, user-facing, metadata-handling functions should be defined in processing.py.

xclim.sdba.adjustment module

Adjustment Methods

`class xclim.sdba.adjustment.BaseAdjustment(`**args*, *_trained=False*, ***kwargs*`)`

Bases: *ParametrizableWithDataset*

Base class for adjustment objects.

Children classes should implement the *train* and / or the *adjust* method.

This base class defined the basic input and output checks. It should only be used for a real adjustment if neither *TrainAdjust* or *Adjust* fit the algorithm.

`_adjust(`**args*, ***kwargs*`)`

`_allow_diff_calendars = True`

`_attribute = '_xclim_adjustment'`

classmethod `_check_inputs`(**inputs*, *group*)

Raise an error if there are chunks along the main dimension.

Also raises if *BaseAdjustment._allow_diff_calendars* is False and calendars differ.

classmethod `_harmonize_units`(**inputs*, *target: str | None = None*)

Convert all inputs to the same units.

If the target unit is not given, the units of the first input are used.

Returns the converted inputs and the target units.

classmethod `_train`(*hist*, ***kwargs*)

class `xclim.sdba.adjustment.DetrendedQuantileMapping`(**args*, *_trained=False*, ***kwargs*)

Bases: `TrainAdjust`

Detrended Quantile Mapping bias-adjustment.

The algorithm follows these steps, 1-3 being the 'train' and 4-6, the 'adjust' steps.

1. A scaling factor that would make the mean of *hist* match the mean of *ref* is computed.
2. *ref* and *hist* are normalized by removing the "dayofyear" mean.
3. Adjustment factors are computed between the quantiles of the normalized *ref* and *hist*.
4. *sim* is corrected by the scaling factor, and either normalized by "dayofyear" and detrended group-wise or directly detrended per "dayofyear", using a linear fit (modifiable).
5. Values of detrended *sim* are matched to the corresponding quantiles of normalized *hist* and corrected accordingly.
6. The trend is put back on the result.

$$F^{-1}_{ref}\left\{F_{hist}\left[\frac{\overline{hist}\cdot sim}{\overline{sim}}\right]\right\}\frac{\overline{sim}}{\overline{hist}}$$

where F is the cumulative distribution function (CDF) and $\overline{xyz}$ is the linear trend of the data. This equation is valid for multiplicative adjustment. Based on the DQM method of [Cannon2015].

Parameters

- **Train step**
- **nquantiles** (*int or 1d array of floats*) – The number of quantiles to use. See *equally_spaced_nodes()*. An array of quantiles [0, 1] can also be passed. Defaults to 20 quantiles.
- **kind** (*{'+', ''}**) – The adjustment kind, either additive or multiplicative. Defaults to "+".
- **group** (*Union[str, Grouper]*) – The grouping information. See *xclim.sdba.base.Grouper* for details. Default is "time", meaning an single adjustment group along dimension "time".
- **Adjust step**
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use when interpolating the adjustment factors. Defaults to "nearest".
- **detrend** (*int or BaseDetrend instance*) – The method to use when detrending. If an int is passed, it is understood as a PolyDetrend (polynomial detrending) degree. Defaults to 1 (linear detrending)

- **extrapolation** (*{'constant', 'nan'}*) – The type of extrapolation to use. See `xclim.sdba.utils.extrapolate_qm()` for details. Defaults to "constant".

References

`_adjust`(*sim*, *interp='nearest'*, *extrapolation='constant'*, *detrend=1*)

`_allow_diff_calendars = False`

`classmethod _train`(*ref: xr.DataArray*, *hist: xr.DataArray*, ***, *nquantiles: int | np.ndarray = 20*, *kind: str = '+'*, *group: str | Grouper = 'time'*)

`class xclim.sdba.adjustment.EmpiricalQuantileMapping`(**args*, *_trained=False*, ***kwargs*)

Bases: `TrainAdjust`

Empirical Quantile Mapping bias-adjustment.

Adjustment factors are computed between the quantiles of *ref* and *sim*. Values of *sim* are matched to the corresponding quantiles of *hist* and corrected accordingly.

$$F^{-1}_{ref}(F_{hist}(sim))$$

where F is the cumulative distribution function (CDF) and *mod* stands for model data.

Parameters

- **Train step**
- **nquantiles** (*int or 1d array of floats*) – The number of quantiles to use. Two endpoints at 1e-6 and 1 - 1e-6 will be added. An array of quantiles [0, 1] can also be passed. Defaults to 20 quantiles.
- **kind** (*{'+', ''}**) – The adjustment kind, either additive or multiplicative. Defaults to "+".
- **group** (*Union[str, Grouper]*) – The grouping information. See *`xclim.sdba.base.Grouper`* for details. Default is "time", meaning an single adjustment group along dimension "time".
- **Adjust step**
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use when interpolating the adjustment factors. Defaults to "nearset".
- **extrapolation** (*{'constant', 'nan'}*) – The type of extrapolation to use. See `xclim.sdba.utils.extrapolate_qm()` for details. Defaults to "constant".

References

Dequé, M. (2007). Frequency of precipitation and temperature extremes over France in an anthropogenic scenario: Model results and statistical correction according to observed values. Global and Planetary Change, 57(1–2), 16–26. https://doi.org/10.1016/j.gloplacha.2006.11.030

_allow_diff_calendars = False

`_adjust`(*sim*, *interp='nearest'*, *extrapolation='constant'*)

`_allow_diff_calendars = False`

classmethod `_train`(*ref: xr.DataArray*, *hist: xr.DataArray*, ***, *nquantiles: int | np.ndarray = 20*, *kind: str = '+'*, *group: str |* Grouper *= 'time'*)

class `xclim.sdba.adjustment.ExtremeValues`(**args*, *_trained=False*, ***kwargs*)

Bases: `TrainAdjust`

Adjustment correction for extreme values.

The tail of the distribution of adjusted data is corrected according to the bias between the parametric Generalized Pareto distributions of the simulatated and reference data, [RRJF2021]. The distributions are composed of the maximal values of clusters of "large" values. With "large" values being those above *cluster_thresh*. Only extreme values, whose quantile within the pool of large values are above *q_thresh*, are re-adjusted. See Notes.

This adjustment method should be considered experimental and used with care.

Parameters

- **Train step**
- **cluster_thresh** (*Quantity (str with units)*) – The threshold value for defining clusters.
- **q_thresh** (*float*) – The quantile of "extreme" values, [0, 1[. Defaults to 0.95.
- **ref_params** (*xr.DataArray, optional*) – Distribution parameters to use instead of fitting a GenPareto distribution on *ref*.
- **Adjust step**
- **scen** (*DataArray*) – This is a second-order adjustment, so the adjust method needs the first-order adjusted timeseries in addition to the raw "sim".
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use when interpolating the adjustment factors. Defaults to "linear".
- **extrapolation** (*{'constant', 'nan'}*) – The type of extrapolation to use. See `extrapolate_qm()` for details. Defaults to "constant".
- **frac** (*float*) – Fraction where the cutoff happens between the original scen and the corrected one. See Notes,]0, 1]. Defaults to 0.25.
- **power** (*float*) – Shape of the correction strength, see Notes. Defaults to 1.0.

Notes

Extreme values are extracted from *ref*, *hist* and *sim* by finding all "clusters", i.e. runs of consecutive values above *cluster_thresh*. The *q_thresh`th percentile of these values is taken on `ref* and *hist* and becomes *thresh*, the extreme value threshold. The maximal value of each cluster, if it exceeds that new threshold, is taken and Generalized Pareto distributions are fitted to them, for both *ref* and *hist*. The probabilities associated with each of these extremes in *hist* is used to find the corresponding value according to *ref*'s distribution. Adjustment factors are computed as the bias between those new extremes and the original ones.

In the adjust step, a Generalized Pareto distributions is fitted on the cluster-maximums of *sim* and it is used to associate a probability to each extreme, values over the *thresh* compute in the training, without the clustering. The adjustment factors are computed by interpolating the trained ones using these probabilities and the probabilities computed from *hist*.

Finally, the adjusted values (C_i) are mixed with the pre-adjusted ones (*scen*, D_i) using the following transition function:

$$V_i = C_i * \tau + D_i * (1 - \tau)$$

Where τ is a function of sim's extreme values (unadjusted, S_i) and of arguments `frac` (f) and `power` (p):

$$\tau = \left(\frac{1}{f}\frac{S - min(S)}{max(S) - min(S)}\right)^p$$

Code based on an internal Matlab source and partly ib the *biascorrect_extremes* function of the julia package [ClimateTools].

Because of limitations imposed by the lazy computing nature of the dask backend, it is not possible to know the number of cluster extremes in *ref* and *hist* at the moment the output data structure is created. This is why the code tries to estimate that number and usually overestimates it. In the training dataset, this translated into a *quantile* dimension that is too large and variables *af* and *px_hist* are assigned NaNs on extra elements. This has no incidence on the calculations themselves but requires more memory than is useful.

References

`_adjust`(*sim: DataArray*, *scen: DataArray*, ***, *frac: float = 0.25*, *power: float = 1.0*, *interp: str = 'linear'*, *extrapolation: str = 'constant'*)

`classmethod _train`(*ref: DataArray*, *hist: DataArray*, ***, *cluster_thresh: str*, *ref_params: Optional[Dataset] = None*, *q_thresh: float = 0.95*)

`class xclim.sdba.adjustment.LOCI`(**args*, *_trained=False*, ***kwargs*)

Bases: `TrainAdjust`

Local Intensity Scaling (LOCI) bias-adjustment.

This bias adjustment method is designed to correct daily precipitation time series by considering wet and dry days separately ([Schmidli2006]).

Multiplicative adjustment factors are computed such that the mean of *hist* matches the mean of *ref* for values above a threshold.

The threshold on the training target *ref* is first mapped to *hist* by finding the quantile in *hist* having the same exceedance probability as thresh in *ref*. The adjustment factor is then given by

$$s = \frac{\langle ref : ref \geq t_{ref} \rangle - t_{ref}}{\langle hist : hist \geq t_{hist} \rangle - t_{hist}}$$

In the case of precipitations, the adjustment factor is the ratio of wet-days intensity.

For an adjustment factor s, the bias-adjustment of *sim* is:

$$sim(t) = \max\left(t_{ref} + s \cdot (hist(t) - t_{hist}), 0\right)$$

Parameters

- **Train step**
- **group** (*Union[str, Grouper]*) – The grouping information. See `xclim.sdba.base.Grouper` for details. Default is "time", meaning an single adjustment group along dimension "time".

- **thresh** (*str*) – The threshold in *ref* above which the values are scaled.
- **Adjust step**
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use then interpolating the adjustment factors. Defaults to "linear".

References

`_adjust`(*sim*, *interp='linear'*)

`_allow_diff_calendars = False`

`classmethod _train`(*ref: xr.DataArray*, *hist: xr.DataArray*, ***, *thresh: str*, *group: str | Grouper = 'time'*)

`class xclim.sdba.adjustment.NpdfTransform`(**args*, *_trained=False*, ***kwargs*)

Bases: `Adjust`

N-dimensional probability density function transform.

This adjustment object combines both training and adjust steps in the *adjust* class method.

A multivariate bias-adjustment algorithm described by [Cannon2018], as part of the MBCn algorithm, based on a color-correction algorithm described by [Pitie2005].

This algorithm in itself, when used with QuantileDeltaMapping, is NOT trend-preserving. The full MBCn algorithm includes a reordering step provided here by `xclim.sdba.processing.reordering()`.

See notes for an explanation of the algorithm.

Parameters

- **base** (*BaseAdjustment*) – An univariate bias-adjustment class. This is untested for anything else than QuantileDeltaMapping.
- **base_kws** (*dict, optional*) – Arguments passed to the training of the univariate adjustment.
- **n_escore** (*int*) – The number of elements to send to the escore function. The default, 0, means all elements are included. Pass -1 to skip computing the escore completely. Small numbers result in less significative scores, but the execution time goes up quickly with large values.
- **n_iter** (*int*) – The number of iterations to perform. Defaults to 20.
- **pts_dim** (*str*) – The name of the "multivariate" dimension. Defaults to "multivar", which is the normal case when using `xclim.sdba.base.stack_variables()`.
- **adj_kws** (*dict, optional*) – Dictionary of arguments to pass to the adjust method of the univariate adjustment.
- **rot_matrices** (*xr.DataArray, optional*) – The rotation matrices as a 3D array ('iterations', <pts_dim>, <anything>), with shape (n_iter, <N>, <N>). If left empty, random rotation matrices will be automatically generated.

Notes

The historical reference (T, for "target"), simulated historical (H) and simulated projected (S) datasets are constructed by stacking the timeseries of N variables together. The algorithm is broken into the following steps:

1. Rotate the datasets in the N-dimensional variable space with $\mathbf{R}$, a random rotation NxN matrix.

..math

```
\tilde{\mathbf{T}} = \mathbf{T}\mathbf{R} \
\tilde{\mathbf{H}} = \mathbf{H}\mathbf{R} \
\tilde{\mathbf{S}} = \mathbf{S}\mathbf{R}
```

2. An univariate bias-adjustment $\mathcal{F}$ is used on the rotated datasets. The adjustments are made in additive mode, for each variable i.

$$\hat{\mathbf{H}}_i, \hat{\mathbf{S}}_i = \mathcal{F}\left(\tilde{\mathbf{T}}_i, \tilde{\mathbf{H}}_i, \tilde{\mathbf{S}}_i\right)$$

3. The bias-adjusted datasets are rotated back.

$$\mathbf{H}' = \hat{\mathbf{H}}\mathbf{R}$$
$$\mathbf{S}' = \hat{\mathbf{S}}\mathbf{R}$$

These three steps are repeated a certain number of times, prescribed by argument `n_iter`. At each iteration, a new random rotation matrix is generated.

The original algorithm ([Pitie2005]), stops the iteration when some distance score converges. Following [Cannon2018] and the MBCn implementation in [CannonR], we instead fix the number of iterations.

As done by [Cannon2018], the distance score chosen is the "Energy distance" from [SkezelyRizzo2004] (see *`xclim.sdba.processing.escore()`*).

The random matrices are generated following a method laid out by [Mezzadri2006].

This is only part of the full MBCn algorithm, see *Statistical Downscaling and Bias-Adjustment* for an example on how to replicate the full method with xclim. This includes a standardization of the simulated data beforehand, an initial univariate adjustment and the reordering of those adjusted series according to the rank structure of the output of this algorithm.

References

`classmethod _adjust`(*ref: xr.DataArray, hist: xr.DataArray, sim: xr.DataArray, *, base: TrainAdjust = <class 'xclim.sdba.adjustment.QuantileDeltaMapping'>, base_kws: Mapping[str, Any] | None = None, n_escore: int = 0, n_iter: int = 20, pts_dim: str = 'multivar', adj_kws: Mapping[str, Any] | None = None, rot_matrices: xr.DataArray | None = None*)

`class xclim.sdba.adjustment.PrincipalComponents`(**args, _trained=False, **kwargs*)

Bases: `TrainAdjust`

Principal component adjustment.

This bias-correction method maps model simulation values to the observation space through principal components ([Hnilica2017]). Values in the simulation space (multiple variables, or multiple sites) can be thought of as coordinates along axes, such as variable, temperature, etc. Principal components (PC) are a linear combinations of the original variables where the coefficients are the eigenvectors of the covariance matrix. Values can then be expressed as coordinates along the PC axes. The method makes the assumption that bias-corrected values have the same coordinates along the PC axes of the observations. By converting from the observation PC space to the original space, we get bias corrected values. See notes for a mathematical explanation.

Note that *principal components* is meant here as the algebraic operation defining a coordinate system based on the eigenvectors, not statistical principal component analysis.

Parameters

- **group** (*Union[str, Grouper]*) – The main dimension and grouping information. See Notes. See `xclim.sdba.base.Grouper` for details. The adjustment will be performed on each group independently. Default is "time", meaning an single adjustment group along dimension "time".
- **best_orientation** (*{'simple', 'full'}*) – Which method to use when searching for the best principal component orientation. See `best_pc_orientation_simple()` and `best_pc_orientation_full()`. "full" is more precise, but it is much slower.
- **crd_dim** (*str*) – The data dimension along which the multiple simulation space dimensions are taken. For a multivariate adjustment, this usually is "multivar", as returned by *sdba.stack_variables*. For a multisite adjustment, this should be the spatial dimension. The training algorithm currently doesn't support any chunking along either *crd_dim*. *group.dim* and *group.add_dims*.

Notes

The input data is understood as a set of N points in a M-dimensional space.

- M is taken along *crd_dim*.
- N is taken along the dimensions given through *group* : (the main *dim* but also, if requested, the *add_dims* and *window*).

The principal components (PC) of *hist* and *ref* are used to defined new coordinate systems, centered on their respective means. The training step creates a matrix defining the transformation from *hist* to *ref*:

$$scen = e_R + \mathbf{T}(sim - e_H)$$

Where:

$$\mathbf{T} = \mathbf{R}\mathbf{H}^{-1}$$

$\mathbf{R}$ is the matrix transforming from the PC coordinates computed on *ref* to the data coordinates. Similarly, $\mathbf{H}$ is transform from the *hist* PC to the data coordinates ($\mathbf{H}$ is the inverse transformation). e_R and e_H are the centroids of the *ref* and *hist* distributions respectively. Upon running the *adjust* step, one may decide to use e_S, the centroid of the *sim* distribution, instead of e_H.

References

`_adjust`(*sim*)

`classmethod _train`(*ref: xr.DataArray*, *hist: xr.DataArray*, ***, *crd_dim: str*, *best_orientation: str = 'simple'*, *group: str | Grouper = 'time'*) `class xclim.sdba.adjustment.QuantileDeltaMapping(*args, _trained=False, **kwargs)`

Bases: `EmpiricalQuantileMapping`

Quantile Delta Mapping bias-adjustment.

Adjustment factors are computed between the quantiles of *ref* and *hist*. Quantiles of *sim* are matched to the corresponding quantiles of *hist* and corrected accordingly.

$$sim\frac{F_{ref}^{-1}\left[F_{sim}(sim)\right]}{F_{hist}^{-1}\left[F_{sim}(sim)\right]}$$

where F is the cumulative distribution function (CDF). This equation is valid for multiplicative adjustment. The algorithm is based on the "QDM" method of [Cannon2015].

Parameters

- **Train step**
- **nquantiles** (*int or 1d array of floats*) – The number of quantiles to use. See `equally_spaced_nodes()`. An array of quantiles [0, 1] can also be passed. Defaults to 20 quantiles.
- **kind** (*{'+', ''}**) – The adjustment kind, either additive or multiplicative. Defaults to "+".
- **group** (*Union[str, Grouper]*) – The grouping information. See `xclim.sdba.base.Grouper` for details. Default is "time", meaning an single adjustment group along dimension "time".
- **Adjust step**
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use when interpolating the adjustment factors. Defaults to "nearest".
- **extrapolation** (*{'constant', 'nan'}*) – The type of extrapolation to use. See `xclim.sdba.utils.extrapolate_qm()` for details. Defaults to "constant".
- **Extra diagnostics**
- ————————
- **In adjustment**
- **quantiles** (The quantile of each value of *sim*. The adjustment factor is interpolated using this as the "quantile" axis on *ds.af*.)

References

`_adjust(sim, interp='nearest', extrapolation='constant')`

`class xclim.sdba.adjustment.Scaling(*args, _trained=False, **kwargs)`

Bases: `TrainAdjust`

Scaling bias-adjustment.

Simple bias-adjustment method scaling variables by an additive or multiplicative factor so that the mean of *hist* matches the mean of *ref*.

Parameters

- **Train step**

- **group** (*Union[str, Grouper]*) – The grouping information. See *xclim.sdba.base.Grouper* for details. Default is "time", meaning an single adjustment group along dimension "time".
- **kind** (*{'+', ''}**) – The adjustment kind, either additive or multiplicative. Defaults to "+".
- **Adjust step**
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use then interpolating the adjustment factors. Defaults to "nearest".

`_adjust`(*sim*, *interp='nearest'*)

`_allow_diff_calendars = False`

`classmethod _train`(*ref: xr.DataArray*, *hist: xr.DataArray*, ***, *group: str | Grouper = 'time'*, *kind: str = '+'*)

xclim.sdba.base module

Base Classes and Developer Tools

`class xclim.sdba.base.Grouper`(*group: str*, *window: int = 1*, *add_dims: Sequence[str] | set[str] | None = None*)

Bases: *Parametrizable*

Grouper inherited class for parameterizable classes.

`ADD_DIMS = '<ADD_DIMS>'`

`DIM = '<DIM>'`

`PROP = '<PROP>'`

`_repr_hide_params = ['dim', 'prop']`

`apply`(*func: FunctionType | str*, *da: xr.DataArray | Mapping[str, xr.DataArray] | xr.Dataset*, *main_only: bool = False*, ***kwargs*)

Apply a function group-wise on DataArrays.

Parameters

- **func** (*Union[FunctionType, str]*) – The function to apply to the groups, either a callable or a *xr.core.groupby.GroupBy* method name as a string. The function will be called as *func(group, dim=dims, **kwargs)*. See *main_only* for the behaviour of *dims*.
- **da** (*Union[xr.DataArray, Mapping[str, xr.DataArray], xr.Dataset]*) – The DataArray on which to apply the function. Multiple arrays can be passed through a dictionary. A dataset will be created before grouping.
- **main_only** (*bool*) – Whether to call the function with the main dimension only (if True) or with all grouping dims (if False, default) (including the window and dimensions given through *add_dims*). The dimensions used are also written in the "group_compute_dims" attribute. If all the input arrays are missing one of the 'add_dims', it is silently omitted.
- **kwargs** – Other keyword arguments to pass to the function.

Returns

DataArray or Dataset – Attributes "group", "group_window" and "group_compute_dims" are added.

If the function did not reduce the array:

- The output is sorted along the main dimension.
- The output is rechunked to match the chunks on the input If multiple inputs with differing chunking were given as inputs, the chunking with the smallest number of chunks is used.

If the function reduces the array:

- If there is only one group, the singleton dimension is squeezed out of the output
- The output is rechunked as to have only 1 chunk along the new dimension.

Notes

For the special case where a Dataset is returned, but only some of its variable where reduced by the grouping, xarray's *GroupBy.map* will broadcast everything back to the ungrouped dimensions. To overcome this issue, function may add a "_group_apply_reshape" attribute set to True on the variables that should be reduced and these will be re-grouped by calling *da.groupby(self.name).first()*.

`property freq`

Format a frequency string corresponding to the group.

For use with xarray's resampling functions.

`classmethod from_kwargs(`***kwargs*`)`

Parameterize groups using kwargs.

`get_coordinate(`*ds=None*`)`

Return the coordinate as in the output of group.apply.

Currently, only implemented for groupings with prop == *month* or *dayofyear*. For prop == *dayfofyear*, a ds (Dataset or DataArray) can be passed to infer the max day of year from the available years and calendar.

`get_index(`*da: xr.DataArray | xr.Dataset*, *interp: bool | None = None*`)`

Return the group index of each element along the main dimension.

Parameters

- **da** (*Union[xr.DataArray, xr.Dataset]*) – The input array/dataset for which the group index is returned. It must have Grouper.dim as a coordinate.
- **interp** (*bool, optional*) – If True, the returned index can be used for interpolation. Only value for month grouping, where integer values represent the middle of the month, all other days are linearly interpolated in between.

Returns

xr.DataArray – The index of each element along *Grouper.dim*. If *Grouper.dim* is *time* and *Grouper.prop* is None, an uniform array of True is returned. If *Grouper.prop* is a time accessor (month, dayofyear, etc), an numerical array is returned, with a special case of *month* and *interp=True*. If *Grouper.dim* is not *time*, the dim is simply returned.

`group`(*da: xr.DataArray | xr.Dataset = None*, *main_only=False*, ***das: xr.DataArray*)

Return a xr.core.groupby.GroupBy object.

More than one array can be combined to a dataset before grouping using the *das* kwargs. A new *window* dimension is added if *self.window* is larger than 1. If *Grouper.dim* is 'time', but 'prop' is None, the whole array is grouped together.

When multiple arrays are passed, some of them can be grouped along the same group as self. They are boadcasted, merged to the grouping dataset and regrouped in the output.

`property prop_name`

Create a significant name for the grouping.

`class xclim.sdba.base.Parametrizable`

Bases: `dict`

Helper base class resembling a dictionary.

This object is _completely_ defined by the content of its internal dictionary, accessible through item access (*self['attr']*) or in *self.parameters*. When serializing and restoring this object, only members of that internal dict are preserved. All other attributes set directly with *self.attr = value* will not be preserved upon serialization and restoration of the object with *[json]pickle*. dictionary. Other variables set with *self.var = data* will be lost in the serialization process. This class is best serialized and restored with *jsonpickle*.

`_repr_hide_params = []`

`property parameters`

All parameters as a dictionary. Read-only.

`class xclim.sdba.base.ParametrizableWithDataset`

Bases: *Parametrizable*

Parametrizeable class that also has a *ds* attribute storing a dataset.

`_attribute = '_xclim_parameters'`

`classmethod from_dataset`(*ds: Dataset*)

Create an instance from a dataset.

The dataset must have a global attribute with a name corresponding to *cls._attribute*, and that attribute must be the result of *jsonpickle.encode(object)* where object is of the same type as this object.

`set_dataset`(*ds: Dataset*)

Store an xarray dataset in the *ds* attribute.

Useful with custom object initialization or if some external processing was performed.

`xclim.sdba.base._decode_cf_coords`(*ds*)

Decode coords in-place.

`xclim.sdba.base.duck_empty`(*dims*, *sizes*, *dtype='float64'*, *chunks=None*)

Return an empty DataArray based on a numpy or dask backend, depending on the chunks argument.

`xclim.sdba.base.map_blocks`(*reduces: Optional[Sequence[str]] = None*, ***outvars*)

Decorator for declaring functions and wrapping them into a map_blocks.

Takes care of constructing the template dataset. Dimension order is not preserved. The decorated function must always have the signature: `func(ds, **kwargs)`, where ds is a DataArray or a Dataset. It must always output a dataset matching the mapping passed to the decorator.

Parameters

- **reduces** (*sequence of strings*) – Name of the dimensions that are removed by the function.
- **outvars** – Mapping from variable names in the output to their *new* dimensions. The placeholders *Grouper.PROP*, *Grouper.DIM* and *Grouper.ADD_DIMS* can be used to signify *group.prop*,`group.dim` and *group.add_dims* respectively. If an output keeps a dimension that another loses, that dimension name must be given in *reduces* and in the list of new dimensions of the first output.

`xclim.sdba.base.map_groups(`*reduces: Optional[Sequence[str]] = None*, *main_only: bool = False*, ***out_vars*`)`

Decorator for declaring functions acting only on groups and wrapping them into a map_blocks.

This is the same as *map_blocks* but adds a call to *group.apply()* in the mapped func and the default value of *reduces* is changed.

The decorated function must have the signature: `func(ds, dim, **kwargs)`. Where ds is a DataAray or Dataset, dim is the group.dim (and add_dims). The *group* argument is stripped from the kwargs, but must evidently be provided in the call.

Parameters

- **reduces** (*sequence of str*) – Dimensions that are removed from the inputs by the function. Defaults to [Grouper.DIM, Grouper.ADD_DIMS] if main_only is False, and [Grouper.DIM] if main_only is True. See `map_blocks()`.
- **main_only** (*bool*) – Same as for `Grouper.apply()`.
- **out_vars** – Mapping from variable names in the output to their *new* dimensions. The placeholders *Grouper.PROP*, *Grouper.DIM* and *Grouper.ADD_DIMS* can be used to signify *group.prop*,`group.dim` and *group.add_dims* respectively. If an output keeps a dimension that another loses, that dimension name must be given in *reduces* and in the list of new dimensions of the first output.

See also:

`map_blocks()`

`xclim.sdba.base.parse_group(`*func: Callable*, *kwargs=None*, *allow_only=None*`)` → Callable

Parse the kwargs given to a function to set the *group* arg with a Grouper object.

This function can be used as a decorator, in which case the parsing and updating of the kwargs is done at call time. It can also be called with a function from which extract the default group and kwargs to update, in which case it returns the updated kwargs.

If allow_only is given, an exception is raised when the parsed group is not within that list.

xclim.sdba.detrending module

Detrending Objects

`class xclim.sdba.detrending.BaseDetrend(`***, *group: Grouper | str = 'time'*, *kind: str = '+'*, ***kwargs*`)`

Bases: `ParametrizableWithDataset`

Base class for detrending objects.

Defines three methods:

fit(da) : Compute trend from da and return a new _fitted_ Detrend object. detrend(da) : Return detrended array. retrend(da) : Puts trend back on da.

A fitted Detrend object is unique to the trend coordinate of the object used in *fit*, (usually 'time'). The computed trend is stored in `Detrend.ds.trend`.

Subclasses should implement `_get_trend_group()` or `_get_trend()`. The first will be called in a `group.apply(..., main_only=True)`, and should return a single DataArray. The second allows the use of functions wrapped in `map_groups()` and should also return a single DataArray.

The subclasses may reimplement `_detrend` and `_retrend`.

`_detrend`(*da*, *trend*)

`_get_trend`(*da: DataArray*)

Compute the trend along the self.group.dim as found on da.

If da is a DataArray (and has a *dtype* attribute), the trend is cast to have the same dtype.

Notes

This method applies *_get_trend_group* with *self.group*.

`_get_trend_group`(*grpd*, ***, *dim*)

`_retrend`(*da*, *trend*)

`detrend`(*da: DataArray*)

Remove the previously fitted trend from a DataArray.

`fit`(*da: DataArray*)

Extract the trend of a DataArray along a specific dimension.

Returns a new object that can be used for detrending and retrending. Fitted objects are unique to the fitted coordinate used.

`property fitted`

Return whether instance is fitted.

`retrend`(*da: DataArray*)

Put the previously fitted trend back on a DataArray.

`class xclim.sdba.detrending.LoessDetrend`(*group='time'*, *kind='+'*, *f=0.2*, *niter=1*, *d=0*, *weights='tricube'*, *equal_spacing=None*, *skipna=True*)

Bases: `BaseDetrend`

Detrend time series using a LOESS regression.

The fit is a piecewise linear regression. For each point, the contribution of all neighbors is weighted by a bell-shaped curve (gaussian) with parameters sigma (std). The x-coordinate of the DataArray is scaled to [0,1] before the regression is computed.

Parameters

- **group** (*Union[str, Grouper]*) – The grouping information. See `xclim.sdba.base.Grouper` for details. The fit is performed along the group's main dim.
- **kind** (*{'*', '+'}*) – The way the trend is removed or added, either additive or multiplicative.
- **d** (*[0, 1]*) – Order of the local regression. Only 0 and 1 currently implemented.

- **f** (*float*) – Parameter controlling the span of the weights, between 0 and 1.
- **niter** (*int*) – Number of robustness iterations to execute.
- **weights** (*["tricube", "gaussian"]*) – Shape of the weighting function: "tricube" : a smooth top-hat like curve, f gives the span of non-zero values. "gaussian" : a gaussian curve, f gives the span for 95% of the values.
- **skipna** (*bool*) – If True (default), missing values are not included in the loess trend computation and thus are not propagated. The output will have the same missing values as the input.

Notes

LOESS smoothing is computationally expensive. As it relies on a loop on gridpoints, it can be useful to use smaller than usual chunks. Moreover, it suffers from heavy boundary effects. As a rule of thumb, the outermost N * f/2 points should be considered dubious. (N is the number of points along each group)

`_get_trend(da)`

Compute the trend along the self.group.dim as found on da.

If da is a DataArray (and has a *dtype* attribute), the trend is cast to have the same dtype.

Notes

This method applies *_get_trend_group* with *self.group*.

`class xclim.sdba.detrending.MeanDetrend(*, group: Grouper | str = 'time', kind: str = '+', **kwargs)`

Bases: `BaseDetrend`

Simple detrending removing only the mean from the data, quite similar to normalizing.

`_get_trend(da)`

Compute the trend along the self.group.dim as found on da.

If da is a DataArray (and has a *dtype* attribute), the trend is cast to have the same dtype.

Notes

This method applies *_get_trend_group* with *self.group*.

`class xclim.sdba.detrending.NoDetrend(*, group: Grouper | str = 'time', kind: str = '+', **kwargs)`

Bases: `BaseDetrend`

Convenience class for polymorphism. Does nothing.

`_detrend(da, trend)`

`_get_trend_group(da, *, dim)`

`_retrend(da, trend)`

`class xclim.sdba.detrending.PolyDetrend(`*group='time', kind='+', degree=4, preserve_mean=False*`)`

Bases: `BaseDetrend`

Detrend time series using a polynomial regression.

Parameters

- **group** (*Union[str, Grouper]*) – The grouping information. See `xclim.sdba.base.Grouper` for details. The fit is performed along the group's main dim.
- **kind** (*{'', '+'}**) – The way the trend is removed or added, either additive or multiplicative.
- **degree** (*int*) – The order of the polynomial to fit.
- **preserve_mean** (*bool*) – Whether to preserve the mean when de/re-trending. If True, the trend has its mean removed before it is used.

`_get_trend(`*da*`)`

Compute the trend along the self.group.dim as found on da.

If da is a DataArray (and has a *dtype* attribute), the trend is cast to have the same dtype.

Notes

This method applies *_get_trend_group* with *self.group*.

`class xclim.sdba.detrending.RollingMeanDetrend(`*group='time', kind='+', win=30, weights=None, min_periods=None*`)`

Bases: `BaseDetrend`

Detrend time series using a rolling mean.

Parameters

- **group** (*Union[str, Grouper]*) – The grouping information. See `xclim.sdba.base.Grouper` for details. The fit is performed along the group's main dim.
- **kind** (*{'', '+'}**) – The way the trend is removed or added, either additive or multiplicative.
- **win** (*int*) – The size of the rolling window. Units are the steps of the grouped data, which means this detrending is best use with either *group='time'* or *group='time.dayofyear'*. Other grouping will have large jumps included within the windows and :py`:class:*LoessDetrend* might offer a better solution.
- **weights** (*sequence of floats, optional*) – Sequence of length *win*. Defaults to None, which means a flat window.
- **min_periods** (*int, optional*) – Minimum number of observations in window required to have a value, otherwise the result is NaN. See `xarray.DataArray.rolling()`. Defaults to None, which sets it equal to *win*. Setting both *weights* and this is not implemented yet.

Notes

As for the *LoessDetrend* detrending, important boundary effects are to be expected.

`_get_trend(`*da*`)`

Compute the trend along the self.group.dim as found on da.

If da is a DataArray (and has a *dtype* attribute), the trend is cast to have the same dtype.

Notes

This method applies *_get_trend_group* with *self.group*.

xclim.sdba.loess module

LOESS Smoothing Module

`xclim.sdba.loess._constant_regression(`*xi*, *x*, *y*, *w*`)`

`xclim.sdba.loess._gaussian_weighting(`*x*`)`

Kernel function for loess with a gaussian shape.

The span f covers 95% of the gaussian.

`xclim.sdba.loess._linear_regression(`*xi*, *x*, *y*, *w*`)`

`xclim.sdba.loess._loess_nb(`*x*, *y*, *f=0.5*, *niter=2*, *weight_func=CPUDispatcher(<function _tricube_weighting>)*, *reg_func=CPUDispatcher(<function _linear_regression>)*, *dx=0*, *skipna=True*`)`

1D Locally weighted regression: fits a nonparametric regression curve to a scatter plot.

The arrays x and y contain an equal number of elements; each pair (x[i], y[i]) defines a data point in the scatter plot. The function returns the estimated (smooth) values of y. Originally proposed in [Cleveland1979].

Users should call *utils.loess_smoothing*. See that function for the main documentation.

Parameters

- **x** (*np.ndarray*) – X-coordinates of the points.
- **y** (*np.ndarray*) – Y-coordinates of the points.
- **f** (*float*) – Parameter controlling the shape of the weight curve. Behavior depends on the weighting function.
- **niter** (*int*) – Number of robustness iterations to execute.
- **weight_func** (*numba func*) – Numba function giving the weights when passed abs(x - xi) / hi
- **dx** (*float*) – The spacing of the x coordinates. If above 0, this enables the optimization for equally spaced x coordinates. Must be 0 if spacing is unequal (default).
- **skipna** (*bool*) – If True (default), remove NaN values before computing the loess. The output has the same missing values as the input.

References

Code adapted from https://gist.github.com/agramfort/850437

`xclim.sdba.loess._tricube_weighting(x)`

Kernel function for loess with a tricubic shape.

`xclim.sdba.loess.loess_smoothing(`*da: xr.DataArray, dim: str = 'time', d: int = 1, f: float = 0.5, niter: int = 2, weights: str | Callable = 'tricube', equal_spacing: bool | None = None, skipna: bool = True*`)`

Locally weighted regression in 1D: fits a nonparametric regression curve to a scatter plot.

Returns a smoothed curve along given dimension. The regression is computed for each point using a subset of neighbouring points as given from evaluating the weighting function locally. Follows the procedure of [Cleveland1979].

Parameters

- **da** (*xr.DataArray*) – The data to smooth using the loess approach.
- **dim** (*str*) – Name of the dimension along which to perform the loess.
- **d** (*[0, 1]*) – Degree of the local regression.
- **f** (*float*) – Parameter controlling the shape of the weight curve. Behavior depends on the weighting function, but it usually represents the span of the weighting function in reference to x-coordinates normalized from 0 to 1.
- **niter** (*int*) – Number of robustness iterations to execute.
- **weights** (*["tricube", "gaussian"] or callable*) – Shape of the weighting function, see notes. The user can provide a function or a string: "tricube" : a smooth top-hat like curve. "gaussian" : a gaussian curve, f gives the span for 95% of the values.
- **equal_spacing** (*bool, optional*) – Whether to use the equal spacing optimization. If *None* (the default), it is activated only if the x-axis is equally-spaced. When activated, *dx = x[1] - x[0]*.
- **skipna** (*bool*) – If True (default), skip missing values (as marked by NaN). The output will have the same missing values as the input.

Notes

As stated in [Cleveland1979], the weighting function $W(x)$ should respect the following conditions:

- $W(x) > 0$ for $|x| < 1$
- $W(-x) = W(x)$
- $W(x)$ is non-increasing for $x \geq 0$
- $W(x) = 0$ for $|x| \geq 0$

If a Callable is provided, it should only accept the 1D *np.ndarray* x which is an absolute value function going from 1 to 0 to 1 around x_i, for all values where $x - x_i < h_i$ with h_i the distance of the rth nearest neighbor of x_i, $r = f * size(x)$.

References

Code adapted from https://gist.github.com/agramfort/850437

xclim.sdba.measures module

Measures Submodule

SDBA diagnostic tests are made up of properties and measures. Measures compare adjusted simulations to a reference, through statistical properties or directly. This framework for the diagnostic tests was inspired by the [VALUE] project.

`xclim.sdba.measures.annual_cycle_correlation(`*sim*, *ref*, *window: int = 15*`)`

Annual cycle correlation.

Pearson correlation coefficient between the smooth day-of-year averaged annual cycles of the simulation and the reference. In the smooth day-of-year averaged annual cycles, each day-of-year is averaged over all years and over a window of days around that day.

Parameters

- **sim** (*xr.DataArray*) – data from the simulation (a time-series for each grid-point)
- **ref** (*xr.DataArray*) – data from the reference (observations) (a time-series for each grid-point)
- **window** (*int*) – Size of window around each day of year around which to take the mean. E.g. If window=31, Jan 1st is averaged over from December 17th to January 16th.

Returns

xr.DataArray, – Annual cycle correlation between the simulation and the reference

`xclim.sdba.measures.bias(`*sim: DataArray*, *ref: DataArray*`)` → DataArray

Bias.

The bias is the simulation minus the reference.

Parameters

- **sim** (*xr.DataArray*) – data from the simulation (one value for each grid-point)
- **ref** (*xr.DataArray*) – data from the reference (observations) (one value for each grid-point)

Returns

xr.DataArray, – Bias between the simulation and the reference

`xclim.sdba.measures.check_same_units_and_convert(`*func*`)` → Callable

Verify that the simulation and the reference have the same units.

If not, it converts the simulation to the units of the reference.

`xclim.sdba.measures.circular_bias(`*sim: DataArray*, *ref: DataArray*`)` → DataArray

Circular bias.

Bias considering circular time series. E.g. The bias between doy 365 and doy 1 is 364, but the circular bias is -1.

Parameters

- **sim** (*xr.DataArray*) – data from the simulation (one value for each grid-point)
- **ref** (*xr.DataArray*) – data from the reference (observations) (one value for each grid-point)

Returns
xr.DataArray, – Circular bias between the simulation and the reference

`xclim.sdba.measures.mae(`*sim: DataArray*, *ref: DataArray*`)` → DataArray

Mean absolute error.

The mean absolute error on the time dimension between the simulation and the reference.

Parameters

- **sim** (*xr.DataArray*) – data from the simulation (a time-series for each grid-point)
- **ref** (*xr.DataArray*) – data from the reference (observations) (a time-series for each grid-point)

Returns
xr.DataArray, – Mean absolute error between the simulation and the reference

`xclim.sdba.measures.ratio(`*sim: DataArray*, *ref: DataArray*`)` → DataArray

Ratio.

The ratio is the quotient of the simulation over the reference.

Parameters

- **sim** (*xr.DataArray*) – data from the simulation (one value for each grid-point)
- **ref** (*xr.DataArray*) – data from the reference (observations) (one value for each grid-point)

Returns
xr.DataArray, – Ratio between the simulation and the reference

`xclim.sdba.measures.relative_bias(`*sim: DataArray*, *ref: DataArray*`)` → DataArray

Relative Bias.

The relative bias is the simulation minus reference, divided by the reference.

Parameters

- **sim** (*xr.DataArray*) – data from the simulation (one value for each grid-point)
- **ref** (*xr.DataArray*) – data from the reference (observations) (one value for each grid-point)

Returns
xr.DataArray, – Relative bias between the simulation and the reference

`xclim.sdba.measures.rmse(`*sim: DataArray*, *ref: DataArray*`)` → DataArray

Root mean square error.

The root mean square error on the time dimension between the simulation and the reference.

Parameters

- **sim** (*xr.DataArray*) – Data from the simulation (a time-series for each grid-point)
- **ref** (*xr.DataArray*) – Data from the reference (observations) (a time-series for each grid-point)

Returns

xr.DataArray, – Root mean square error between the simulation and the reference

xclim.sdba.nbutils module

Numba-accelerated utilities

`xclim.sdba.nbutils._autocorrelation(`*X*`)`

Mean of the NxN pairwise distances of points in X of shape KxN.

Similar to scipy.spatial.distance.pdist(..., 'euclidean')

`xclim.sdba.nbutils._correlation(`*X*, *Y*`)`

Compute a correlation as the mean of pairwise distances between points in X and Y.

X is KxN and Y is KxM, the result is the mean of the MxN distances. Similar to scipy.spatial.distance.cdist(X, Y, 'euclidean')

`xclim.sdba.nbutils._euclidean_norm(`*v*`)`

Compute the euclidean norm of vector v.

`xclim.sdba.nbutils._extrapolate_on_quantiles(`*interp*, *oldx*, *oldg*, *oldy*, *newx*, *newg*, *method='constant'*`)`

Apply extrapolation to the output of interpolation on quantiles with a given grouping.

Arguments are the same as _interp_on_quantiles_2D.

`xclim.sdba.nbutils._first_and_last_nonnull(`*arr*`)`

For each row of arr, get the first and last non NaN elements.

`xclim.sdba.nbutils._quantile(`*arr*, *q*`)`

`xclim.sdba.nbutils.quantile(`*da*, *q*, *dim*`)`

Compute the quantiles from a fixed list *q*.

`xclim.sdba.nbutils.remove_NaNs(`*x*`)`

Remove NaN values from series.

`xclim.sdba.nbutils.vecquantiles(`*da*, *rnk*, *dim*`)`

For when the quantile (rnk) is different for each point.

da and rnk must share all dimensions but dim.

xclim.sdba.processing module

Pre and post processing

`xclim.sdba.processing._get_number_of_elements_by_year(`*time*`)`

Get the number of elements in time in a year by inferring its sampling frequency.

Only calendar with uniform year lengths are supported : 360_day, noleap, all_leap.

`xclim.sdba.processing.adapt_freq`(*ref: xr.DataArray*, *sim: xr.DataArray*, ***, *group: Grouper | str*, *thresh: str = '0 mm d-1'*) → xr.Dataset

Adapt frequency of values under thresh of *sim*, in order to match ref.

This is useful when the dry-day frequency in the simulations is higher than in the references. This function will create new non-null values for *sim/hist*, so that adjustment factors are less wet-biased. Based on [Themessl2012].

Parameters

- **ds** (*xr.Dataset*) – With variables : "ref", Target/reference data, usually observed data. and "sim", Simulated data.
- **dim** (*str*) – Dimension name.
- **group** (*Union[str, Grouper]*) – Grouping information, see base.Grouper
- **thresh** (*str*) – Threshold below which values are considered zero, a quantity with units.

Returns

- **sim_adj** (*xr.DataArray*) – Simulated data with the same frequency of values under threshold than ref. Adjustment is made group-wise.
- **pth** (*xr.DataArray*) – For each group, the smallest value of sim that was not frequency-adjusted. All values smaller were either left as zero values or given a random value between thresh and pth. NaN where frequency adaptation wasn't needed.
- **dP0** (*xr.DataArray*) – For each group, the percentage of values that were corrected in sim.

Notes

With P_0^r the frequency of values under threshold T_0 in the reference (ref) and P_0^s the same for the simulated values,
$DeltaP_0 =$
$fracP_0^s - P_0^r P_0^s$, when positive, represents the proportion of values under T_0 that need to be corrected.

The correction replaces a proportion
$DeltaP_0$ of the values under T_0 in sim by a uniform random number between T_0 and P_{th}, where $P_{th} = F_{ref}^{-1}(F_{sim}(T_0))$ and *F(x)* is the empirical cumulative distribution function (CDF).

References

`xclim.sdba.processing.construct_moving_yearly_window`(*da: Dataset*, *window: int = 21*, *step: int = 1*, *dim: str = 'movingwin'*)

Construct a moving window DataArray.

Stacks windows of *da* in a new 'movingwin' dimension. Windows are always made of full years, so calendar with non-uniform year lengths are not supported.

Windows are constructed starting at the beginning of *da*, if number of given years is not a multiple of *step*, then the last year(s) will be missing as a supplementary window would be incomplete.

Parameters

- **da** (*xr.Dataset*) – A DataArray with a *time* dimension.

- **window** (*int*) – The length of the moving window as a number of years.
- **step** (*int*) – The step between each window as a number of years.
- **dim** (*str*) – The new dimension name. If given, must also be given to *unpack_moving_yearly_window*.

Returns
xr.DataArray – A DataArray with a new *movingwin* dimension and a *time* dimension with a length of 1 window. This assumes downstream algorithms do not make use of the _absolute_ year of the data. The correct timeseries can be reconstructed with `unpack_moving_yearly_window()`. The coordinates of *movingwin* are the first date of the windows.

`xclim.sdba.processing.escore`(*tgt: DataArray*, *sim: DataArray*, *dims: Sequence[str] = ('variables', 'time')*, *N: int = 0*, *scale: bool = False*) → DataArray

Energy score, or energy dissimilarity metric, based on [SzekelyRizzo] and [Cannon18].

Parameters

- **tgt** (*xr.DataArray*) – Target observations.
- **sim** (*xr.DataArray*) – Candidate observations. Must have the same dimensions as *tgt*.
- **dims** (*sequence of 2 strings*) – The name of the dimensions along which the variables and observation points are listed. *tgt* and *sim* can have different length along the second one, but must be equal along the first one. The result will keep all other dimensions.
- **N** (*int*) – If larger than 0, the number of observations to use in the score computation. The points are taken evenly distributed along *obs_dim*.
- **scale** (*bool*) – Whether to scale the data before computing the score. If True, both arrays as scaled according to the mean and standard deviation of *tgt* along *obs_dim*. (std computed with *ddof=1* and both statistics excluding NaN values).

Returns
xr.DataArray – e-score with dimensions not in *dims*.

Notes

Explanation adapted from the "energy" R package documentation. The e-distance between two clusters C_i, C_j (tgt and sim) of size n_i, n_j proposed by Székely and Rizzo (2004) is defined by:

$$e(C_i, C_j) = \frac{1}{2}\frac{n_i n_j}{n_i + n_j}\left[2M_{ij} M_{ii} M_{jj}\right]$$

where

$$M_{ij} = \frac{1}{n_i n_j}\sum_{p=1}^{n_i}\sum_{q=1}^{n_j}\left\|X_{ip} Xjq\right\|.$$

$\|\cdot\|$ denotes Euclidean norm, X_{ip} denotes the p-th observation in the i-th cluster.

The input scaling and the factor $\frac{1}{2}$ in the first equation are additions of [Cannon18] to the metric. With that factor, the test becomes identical to the one defined by [BaringhausFranz]. This version is tested against values taken from Alex Cannon's MBC R package.

References

`xclim.sdba.processing.from_additive_space(`*data: DataArray, lower_bound: Optional[str] = None, upper_bound: Optional[str] = None, trans: Optional[str] = None, units: Optional[str] = None*`)`

Transform back to the physical space a variable that was transformed with *to_addtitive_space*.

Based on [AlavoineGrenier]. If parameters are not present on the attributes of the data, they must be all given are arguments.

Parameters

- **data** (*xr.DataArray*) – A variable that was transform by `to_additive_space()`.
- **lower_bound** (*str, optional*) – The smallest physical value of the variable, as a Quantity string. The final data will have no value smaller or equal to this bound. If None (default), the *sdba_transform_lower* attribute is looked up on *data*.
- **upper_bound** (*str, optional*) – The largest physical value of the variable, as a Quantity string. Only relevant for the logit transformation. The final data will have no value larger or equal to this bound. If None (default), the *sdba_transform_upper* attribute is looked up on *data*.
- **trans** (*{'log', 'logit'}, optional*) – The transformation to use. See notes. If None (the default), the *sdba_transform* attribute is looked up on *data*.
- **units** (*str, optional*) – The units of the data before transformation to the additive space. If None (the default), the *sdba_transform_units* attribute is looked up on *data*.

Returns

xr.DataArray – The physical variable. Attributes are conserved, even if some might be incorrect. Except units which are taken from *sdba_transform_units* if available. All *sdba_transform** attributes are deleted.

Notes

Given a variable that is not usable in an additive adjustment, `to_additive_space()` applied a transformation to a space where additive methods are sensible. Given Y the transformed variable, b_- the lower physical bound of that variable and b_+ the upper physical bound, two back-transformations are currently implemented to get X, the physical variable.

- *log*

$$X = e^Y + b_-$$

- *logit*

$$X' = \frac{1}{1 + e^{-Y}} X = X * (b_+ - b_-) + b_-$$

See also:

`to_additive_space`
for the original transformation.

References

`xclim.sdba.processing.jitter(`*x: xr.DataArray*, *lower: str | None = None*, *upper: str | None = None*, *minimum: str | None = None*, *maximum: str | None = None*`)` → xr.DataArray

Replace values under a threshold and values above another by a uniform random noise.

Not to be confused with R's *jitter*, which adds uniform noise instead of replacing values.

Parameters

- **x** (*xr.DataArray*) – Values.
- **lower** (*str, optional*) – Threshold under which to add uniform random noise to values, a quantity with units. If None, no jittering is performed on the lower end.
- **upper** (*str, optional*) – Threshold over which to add uniform random noise to values, a quantity with units. If None, no jittering is performed on the upper end.
- **minimum** (*str, optional*) – Lower limit (excluded) for the lower end random noise, a quantity with units. If None but *lower* is not None, 0 is used.
- **maximum** (*str, optional*) – Upper limit (excluded) for the upper end random noise, a quantity with units. If *upper* is not None, it must be given.

Returns

xr.DataArray – Same as *x* but values < lower are replaced by a uniform noise in range (minimum, lower) and values >= upper are replaced by a uniform noise in range [upper, maximum). The two noise distributions are independent.

`xclim.sdba.processing.jitter_over_thresh(`*x: DataArray*, *thresh: str*, *upper_bnd: str*`)` → DataArray

Replace values greater than threshold by a uniform random noise.

Do not confuse with R's jitter, which adds uniform noise instead of replacing values.

Parameters

- **x** (*xr.DataArray*) – Values.
- **thresh** (*str*) – Threshold over which to add uniform random noise to values, a quantity with units.
- **upper_bnd** (*str*) – Maximum possible value for the random noise, a quantity with units.

Returns

xr.DataArray

Notes

If thresh is low, this will change the mean value of x.

`xclim.sdba.processing.jitter_under_thresh(`*x: DataArray*, *thresh: str*`)` → DataArray

Replace values smaller than threshold by a uniform random noise.

Do not confuse with R's jitter, which adds uniform noise instead of replacing values.

Parameters

- **x** (*xr.DataArray*) – Values.

- **thresh** (*str*) – Threshold under which to add uniform random noise to values, a quantity with units.

Returns
xr.DataArray

Notes

If thresh is high, this will change the mean value of x.

`xclim.sdba.processing.normalize(`*data: xr.DataArray*, *norm: xr.DataArray | None = None*, ***, *group: Grouper | str*, *kind: str = '+'*`)` → xr.Dataset

Normalize an array by removing its mean.

Normalization if performed group-wise and according to *kind*.

Parameters

- **data** (*xr.DataArray*) – The variable to normalize.
- **norm** (*xr.DataArray, optional*) – If present, it is used instead of computing the norm again.
- **group** (*Union[str, Grouper]*) – Grouping information. See `xclim.sdba.base.Grouper` for details..
- **kind** (*{'+', ''}**) – If *kind* is "+", the mean is subtracted from the mean and if it is '*', it is divided from the data.

Returns

- *xr.DataArray* – Groupwise anomaly.
- **norm** (*xr.DataArray*) – Mean over each group.

`xclim.sdba.processing.reordering(`*ref: DataArray*, *sim: DataArray*, *group: str = 'time'*`)` → Dataset

Reorders data in *sim* following the order of ref.

The rank structure of *ref* is used to reorder the elements of *sim* along dimension "time", optionally doing the operation group-wise.

Parameters

- **sim** (*xr.DataArray*) – Array to reorder.
- **ref** (*xr.DataArray*) – Array whose rank order sim should replicate.
- **group** (*str*) – Grouping information. See `xclim.sdba.base.Grouper` for details.

Returns

- *xr.Dataset* – sim reordered according to ref's rank order.
- *Reference*
- *———*
- **.. [Cannon18] Cannon, A. J. (2018). Multivariate quantile mapping bias correction** (*An N-dimensional probability density function transform for climate model simulations of multiple variables. Climate Dynamics, 50(1), 31–49. https://doi.org/10.1007/s00382-017-3580-6*)

`xclim.sdba.processing.stack_variables(`*ds: Dataset*, *rechunk: bool = True*, *dim: str = 'multivar'*`)`

Stack different variables of a dataset into a single DataArray with a new "variables" dimension.

Variable attributes are all added as lists of attributes to the new coordinate, prefixed with "_". Variables are concatenated in the new dimension in alphabetical order, to ensure coherent behaviour with different datasets.

Parameters

- **ds** (*xr.Dataset*) – Input dataset.
- **rechunk** (*bool*) – If True (default), dask arrays are rechunked with *variables : -1*.
- **dim** (*str*) – Name of dimension along which variables are indexed.

Returns

xr.DataArray – The transformed variable. Attributes are conserved, even if some might be incorrect. Except units, which are replaced with "". Old units are stored in *sdba_transformation_units*. A *sdba_transform* attribute is added, set to the transformation method. *sdba_transform_lower* and *sdba_transform_upper* are also set if the requested bounds are different from the defaults.

Array with variables stacked along *dim* dimension. Units are set to "".

`xclim.sdba.processing.standardize(`*da: xr.DataArray*, *mean: xr.DataArray | None = None*, *std: xr.DataArray | None = None*, *dim: str = 'time'*`)` → tuple[xr.DataArray | xr.Dataset, xr.DataArray, xr.DataArray]

Standardize a DataArray by centering its mean and scaling it by its standard deviation.

Either of both of mean and std can be provided if need be.

Returns the standardized data, the mean and the standard deviation.

`xclim.sdba.processing.to_additive_space(`*data: DataArray*, *lower_bound: str*, *upper_bound: Optional[str] = None*, *trans: str = 'log'*`)`

Transform a non-additive variable into an additive space by the means of a log or logit transformation.

Based on [AlavoineGrenier].

Parameters

- **data** (*xr.DataArray*) – A variable that can't usually be bias-adusted by additive methods.
- **lower_bound** (*str*) – The smallest physical value of the variable, excluded, as a Quantity string. The data should only have values strictly larger than this bound.
- **upper_bound** (*str, optional*) – The largest physical value of the variable, excluded, as a Quantity string. Only relevant for the logit transformation. The data should only have values strictly smaller than this bound.
- **trans** (*{'log', 'logit'}*) – The transformation to use. See notes.

Notes

Given a variable that is not usable in an additive adjustment, this apply a transformation to a space where additive methods are sensible. Given X the variable, b_- the lower physical bound of that variable and b_+ the upper physical bound, two transformations are currently implemented to get Y, the additive-ready variable. ln is the natural logarithm.

- *log*

  $$Y = \ln\left(X - b_-\right)$$

  Usually used for variables with only a lower bound, like precipitation (pr, $prsn$, etc) and daily temperature range (dtr). Both have a lower bound of 0.

- *logit*

  $$X' = (X - b_-)/(b_+ - b_-)Y = \ln\left(\frac{X'}{1 - X'}\right)$$

  Usually used for variables with both a lower and a upper bound, like relative and specific humidity, cloud cover fraction, etc.

This will thus produce *Infinity* and *NaN* values where $X == b_-$ or $X == b_+$. We recommend using `jitter_under_thresh()` and `jitter_over_thresh()` to remove those issues.

See also:

`from_additive_space`
: for the inverse transformation.

`jitter_under_thresh`
: Remove values exactly equal to the lower bound.

`jitter_over_thresh`
: Remove values exactly equal to the upper bound.

References

`xclim.sdba.processing.uniform_noise_like(`*da: DataArray*, *low: float = 1e-06*, *high: float = 0.001*`)` → DataArray

Return a uniform noise array of the same shape as da.

Noise is uniformly distributed between low and high. Alternative method to *jitter_under_thresh* for avoiding zeroes.

`xclim.sdba.processing.unpack_moving_yearly_window(`*da: DataArray*, *dim: str = 'movingwin'*, *append_ends: bool = True*`)`

Unpack a constructed moving window dataset to a normal timeseries, only keeping the central data.

Unpack DataArrays created with `construct_moving_yearly_window()` and recreate a timeseries data. If *append_ends* is False, only keeps the central non-overlapping years. The final timeseries will be (window - step) years shorter than the initial one. If *append_ends* is True, the time points from first and last windows will be included in the final timeseries.

The time points that are not in a window will never be included in the final timeseries. The window length and window step are inferred from the coordinates.

Parameters

- **da** (*xr.DataArray*) – As constructed by `construct_moving_yearly_window()`.
- **dim** (*str*) – The window dimension name as given to the construction function.
- **append_ends** (*bool*) – Whether to append the ends of the timeseries If False, the final timeseries will be (window - step) years shorter than the initial one, but all windows will contribute equally. If True, the year before the middle years of the first window and the years after the middle years of the last window are appended to the middle years. The final timeseries will be the same length as the initial timeseries if the windows span the whole timeseries. The time steps that are not in a window will be left out of the final timeseries.

`xclim.sdba.processing.unstack_variables(`*da: DataArray*, *dim: Optional[str] = None*`)`

Unstack a DataArray created by *stack_variables* to a dataset.

Parameters

- **da** (*xr.DataArray*) – Array holding different variables along *dim* dimension.
- **dim** (*str*) – Name of dimension along which the variables are stacked. If not specified (default), *dim* is inferred from attributes of the coordinate.

Returns

xr.Dataset – Dataset holding each variable in an individual DataArray.

`xclim.sdba.processing.unstandardize(`*da: DataArray*, *mean: DataArray*, *std: DataArray*`)`

Rescale a standardized array by performing the inverse operation of *standardize*.

xclim.sdba.properties module

Properties Submodule

SDBA diagnostic tests are made up of statistical properties and measures. Properties are calculated on both simulation and reference datasets. They collapse the time dimension to one value.

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

`xclim.sdba.properties.STATISTICAL_PROPERTIES: dict[str, Callable] = {'acf': <function acf>, 'annual_cycle_amplitude': <function annual_cycle_amplitude>, 'annual_cycle_phase': <function annual_cycle_phase>, 'corr_btw_var': <function corr_btw_var>, 'mean': <function mean>, 'quantile': <function quantile>, 'relative_frequency': <function relative_frequency>, 'return_value': <function return_value>, 'skewness': <function skewness>, 'spell_length_distribution': <function spell_length_distribution>, 'trend': <function trend>, 'var': <function var>}`

Dictionary of all the statistical properties available.

`xclim.sdba.properties.acf(`*da: xr.DataArray*, ***, *lag: int = 1*, *group: str | Grouper = 'time.season'*`)` → xr.DataArray

Autocorrelation function.

Autocorrelation with a lag over a time resolution and averaged over all years.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **lag** (*int*) – Lag.

- **group** (*{'time.season', 'time.month'}*) – Grouping of the output. E.g. If 'time.month', the autocorrelation is calculated over each month separately for all years. Then, the autocorrelation for all Jan/Feb/. . . is averaged over all years, giving 12 outputs for each grid point.

Returns
xr.DataArray – lag-{lag} autocorrelation of the variable over a {group.prop} and averaged over all years.

See also:

`statsmodels.tsa.stattools.acf`

References

Alavoine M., and Grenier P. (under review) The distinct problems of physical inconsistency and of multivariate bias potentially involved in the statistical adjustment of climate simulations. International Journal of Climatology, submitted on September 19th 2021. (Preprint: https://doi.org/10.31223/X5C34C)

Examples

```
>>> from xclim.testing import open_dataset
>>> pr = open_dataset(path_to_pr_file).pr
>>> acf(da=pr, lag=3, group="time.season")
```

`xclim.sdba.properties.annual_cycle_amplitude`(*da: xr.DataArray*, ***, *amplitude_type: str = 'absolute'*, *group: str | Grouper = 'time'*) → xr.DataArray

Annual cycle amplitude.

The amplitudes of the annual cycle are calculated for each year, then averaged over the all years.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **amplitude_type** (*{'absolute','relative'}*) – Type of amplitude. 'absolute' is the peak-to-peak amplitude. (max - min). 'relative' is a relative percentage. 100 * (max - min) / mean (Recommended for precipitation).

Returns
xr.DataArray – {amplitude_type} amplitude of the annual cycle.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> annual_cycle_amplitude(da=pr, amplitude_type="relative")
```

`xclim.sdba.properties.annual_cycle_phase`(*da: xr.DataArray*, ***, *group: str | Grouper = 'time'*) → xr.DataArray

Annual cycle phase.

The phases of the annual cycle are calculated for each year, then averaged over the all years.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **group** (*{"time", 'time.season', 'time.month'}*) – Grouping of the output. Default: "time".

Returns
xr.DataArray – Phase of the annual cycle. The position (day-of-year) of the maximal value.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> annual_cycle_phase(da=pr)
```

`xclim.sdba.properties.corr_btw_var(`*da1: xr.DataArray, da2: xr.DataArray, *, corr_type: str = 'Spearman', group: str | Grouper = 'time', output: str = 'correlation'*`)` → xr.DataArray

Correlation between two variables.

Spearman or Pearson correlation coefficient between two variables at the time resolution.

Parameters

- **da1** (*xr.DataArray*) – First variable on which to calculate the diagnostic.
- **da2** (*xr.DataArray*) – Second variable on which to calculate the diagnostic.
- **corr_type** (*{'Pearson','Spearman'}*) – Type of correlation to calculate.
- **output** (*{'correlation', 'pvalue'}*) – Wheter to return the correlation coefficient or the p-value.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. Eg. For 'time.month', the correlation would be calculated on each month separately, but with all the years together.

Returns
xr.DataArray – {corr_type} correlation coefficient

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> tasmax = open_dataset("NRCANdaily/nrcan_canada_daily_tasmax_1990.nc").tasmax
>>> corr_btw_var(da1=pr, da2=tasmax, group="time.season")
```

`xclim.sdba.properties.mean(`*da: xr.DataArray, *, group: str | Grouper = 'time'*`)` → xr.DataArray

Mean.

Mean over all years at the time resolution.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. E.g. If 'time.month', the temporal average is performed separately for each month.

Returns
xr.DataArray, – Mean of the variable.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> mean(da=pr, group="time.season")
```

`xclim.sdba.properties.quantile(`*da: xr.DataArray*, ***, *q: float = 0.98*, *group: str | Grouper = 'time'*`)` → xr.DataArray

Quantile.

Returns the quantile q of the distribution of the variable over all years at the time resolution.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **q** (*float*) – Quantile to be calculated. Should be between 0 and 1.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. E.g. If 'time.month', the quantile is computed separately for each month.

Returns

xr.DataArray – Quantile {q} of the variable.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> quantile(da=pr, q=0.9, group="time.season")
```

`xclim.sdba.properties.register_statistical_properties(`*aspect: str*, *seasonal: bool*, *annual: bool*`)` → Callable

Register statistical properties in the STATISTICAL_PROPERTIES dictionary with its aspect and time resolutions.

`xclim.sdba.properties.relative_frequency(`*da: xr.DataArray*, ***, *op: str = '>='*, *thresh: str = '1mm d-1'*, *group: str | Grouper = 'time'*`)` → xr.DataArray

Relative Frequency.

Relative Frequency of days with variable respecting a condition (defined by an operation and a threshold) at the time resolution. The relative freqency is the number of days that satisfy the condition divided by the total number of days.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **op** (*{">", "<", ">=", "<="}*) – Operation to verify the condition. The condition is variable {op} threshold.
- **thresh** (*str*) – Threshold on which to evaluate the condition.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping on the output. Eg. For 'time.month', the relative frequency would be calculated on each month, with all years included.

Returns

xr.DataArray – Relative frequency of the variable.

Examples

```
>>> tasmax = open_dataset(path_to_tasmax_file).tasmax
>>> relative_frequency(da=tasmax, op="<", thresh="0 degC", group="time.season")
```

`xclim.sdba.properties.return_value(`*da: xr.DataArray*, ***, *period: int = 20*, *op: str = 'max'*, *method: str = 'ML'*, *group: str | Grouper = 'time'*`)` → xr.DataArray

Return value.

Return the value corresponding to a return period. On average, the return value will be exceeded (or not exceed for op='min') every return period (eg. 20 years). The return value is computed by first extracting the variable annual maxima/minima, fitting a statistical distribution to the maxima/minima, then estimating the percentile associated with the return period (eg. 95th percentile (1/20) for 20 years)

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **period** (*int*) – Return period. Number of years over which to check if the value is exceeded (or not for op='min').
- **op** (*{'max','min'}*) – Whether we are looking for a probability of exceedance ('max', right side of the distribution) or a probability of non-exceedance (min, left side of the distribution).
- **method** (*{"ML", "PWM"}*) – Fitting method, either maximum likelihood (ML) or probability weighted moments (PWM), also called L-Moments. The PWM method is usually more robust to outliers. However, it requires the lmoments3 libraryto be installed from the *develop* branch. `pip install git+https://github.com/OpenHydrology/lmoments3.git@develop#egg=lmoments3`
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. A distribution of the extremums is done for each group.

Returns

xr.DataArray – {period}-{group} {op} return level of the variable.

Examples

```
>>> tas = open_dataset(path_to_tas_file).tas
>>> return_value(da=tas, group="time.season")
```

`xclim.sdba.properties.skewness(`*da: xr.DataArray*, ***, *group: str | Grouper = 'time'*`)` → xr.DataArray

Skewness.

Skewness of the distribution of the variable over all years at the time resolution.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. E.g. If 'time.month', the skewness is performed separately for each month.

Returns

xr.DataArray – Skewness of the variable.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> skewness(da=pr, group="time.season")
```

See also:

`scipy.stats.skew`

`xclim.sdba.properties.spell_length_distribution(`*da: xr.DataArray, *, method: str = 'amount', op: str = '>=', thresh: str | float = '1 mm d-1', stat: str = 'mean', group: str |* Grouper *= 'time'*) → xr.DataArray

Spell length distribution.

Statistic of spell length distribution when the variable respects a condition (defined by an operation, a method and
a threshold).

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **method** (*{'amount', 'quantile'}*) – Method to choose the threshold. 'amount': The threshold is directly the quantity in {thresh}. It needs to have the same units as {da}. 'quantile': The threshold is calculated as the quantile {thresh} of the distribution.
- **op** (*{">", "<", ">=", "<="}*) – Operation to verify the condition for a spell. The condition for a spell is variable {op} threshold.
- **thresh** (*str or float*) – Threshold on which to evaluate the condition to have a spell. Str with units if the method is "amount". Float of the quantile if the method is "quantile".
- **stat** (*{'mean','max','min'}*) – Statistics to apply to the resampled input at the {group} (e.g. 1-31 Jan 1980) and then over all years (e.g. Jan 1980-2010)
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. E.g. If 'time.month', the spell lengths are coputed separately for each month.

Returns

xr.DataArray – {stat} of spell length distribution when the variable is {op} the {method} {thresh}.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> spell_length_distribution(da=pr, op="<", thresh="1mm d-1", group="time.season")
```

`xclim.sdba.properties.trend(`*da: xr.DataArray, *, group: str |* Grouper *= 'time', output: str = 'slope'*) → xr.DataArray

Linear Trend.

The data is averaged over each time resolution and the interannual trend is returned.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.

- **output** (*{'slope', 'pvalue'}*) – Attributes of the linear regression to return. 'slope' is the slope of the regression line. 'pvalue' is for a hypothesis test whose null hypothesis is that the slope is zero, using Wald Test with t-distribution of the test statistic.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping on the output.

Returns

xr.DataArray – Trend of the variable.

See also:

`scipy.stats.linregress`, `numpy.polyfit`

Examples

```
>>> tas = open_dataset(path_to_tas_file).tas
>>> trend(da=tas, group="time.season")
```

`xclim.sdba.properties.var`(*da: xr.DataArray*, ***, *group: str | Grouper = 'time'*) → xr.DataArray

Variance.

Variance of the variable over all years at the time resolution.

Parameters

- **da** (*xr.DataArray*) – Variable on which to calculate the diagnostic.
- **group** (*{'time', 'time.season', 'time.month'}*) – Grouping of the output. E.g. If 'time.month', the variance is performed separately for each month.

Returns

xr.DataArray – Variance of the variable.

Examples

```
>>> pr = open_dataset(path_to_pr_file).pr
>>> var(da=pr, group="time.season")
```

xclim.sdba.utils module

Statistical Downscaling and Bias Adjustment Utilities

`xclim.sdba.utils._ecdf_1d`(*x*, *value*)

`xclim.sdba.utils._interp_on_quantiles_1D`(*newx*, *oldx*, *oldy*, *method*, *extrap*)

`xclim.sdba.utils._interp_on_quantiles_2D`(*newx*, *newg*, *oldx*, *oldy*, *oldg*, *method*, *extrap*)

`xclim.sdba.utils.add_cyclic_bounds`(*da: xr.DataArray*, *att: str*, *cyclic_coords: bool = True*) → xr.DataArray | xr.Dataset

Reindex an array to include the last slice at the beginning and the first at the end.

This is done to allow interpolation near the end-points.

Parameters

- **da** (*Union[xr.DataArray, xr.Dataset]*) – An array
- **att** (*str*) – The name of the coordinate to make cyclic
- **cyclic_coords** (*bool*) – If True, the coordinates are made cyclic as well, if False, the new values are guessed using the same step as their neighbour.

Returns
Union[xr.DataArray, xr.Dataset] – da but with the last element along att prepended and the last one appended.

`xclim.sdba.utils.apply_correction(`*x: xr.DataArray*, *factor: xr.DataArray*, *kind: str | None = None*`)` → xr.DataArray

Apply the additive or multiplicative correction/adjustment factors.

If kind is not given, default to the one stored in the "kind" attribute of factor.

`xclim.sdba.utils.best_pc_orientation_full(`*R: ndarray*, *Hinv: ndarray*, *Rmean: ndarray*, *Hmean: ndarray*, *hist: ndarray*`)` → ndarray

Return best orientation vector for A according to the method of Alavoine et al. (2021, preprint).

Eigenvectors returned by *pc_matrix* do not have a defined orientation. Given an inverse transform Hinv, a transform R, the actual and target origins Hmean and Rmean and the matrix of training observations hist, this computes a scenario for all possible orientations and return the orientation that maximizes the Spearman correlation coefficient of all variables. The correlation is computed for each variable individually, then averaged.

This trick is explained in [alavoine2021]. See documentation of `sdba.adjustment.PrincipalComponentAdjustment()`.

Parameters

- **R** (*np.ndarray*) – MxM Matrix defining the final transformation.
- **Hinv** (*np.ndarray*) – MxM Matrix defining the (inverse) first transformation.
- **Rmean** (*np.ndarray*) – M vector defining the target distribution center point.
- **Hmean** (*np.ndarray*) – M vector defining the original distribution center point.
- **hist** (*np.ndarray*) – MxN matrix of all training observations of the M variables/sites.

Returns
np.ndarray – M vector of orientation correction (1 or -1).

References

`xclim.sdba.utils.best_pc_orientation_simple(`*R: ndarray*, *Hinv: ndarray*, *val: float = 1000*`)` → ndarray

Return best orientation vector according to a simple test.

Eigenvectors returned by *pc_matrix* do not have a defined orientation. Given an inverse transform Hinv and a transform R, this returns the orientation minimizing the projected distance for a test point far from the origin.

This trick is inspired by the one exposed in [hnilica2017]. For each possible orientation vector, the test point is reprojected and the distance from the original point is computed. The orientation minimizing that distance is chosen. See documentation of *sdba.adjustment.PrincipalComponentAdjustment*.

Parameters

- **R** (*np.ndarray*) – MxM Matrix defining the final transformation.

- **Hinv** (*np.ndarray*) – MxM Matrix defining the (inverse) first transformation.
- **val** (*float*) – The coordinate of the test point (same for all axes). It should be much greater than the largest furthest point in the array used to define B.

Returns
np.ndarray – Mx1 vector of orientation correction (1 or -1).

References

`xclim.sdba.utils.broadcast(`*grouped: xr.DataArray, x: xr.DataArray, *, group: str | Grouper = 'time', interp: str = 'nearest', sel: Mapping[str, xr.DataArray] | None = None*`)` → xr.DataArray

Broadcast a grouped array back to the same shape as a given array.

Parameters

- **grouped** (*xr.DataArray*) – The grouped array to broadcast like *x*.
- **x** (*xr.DataArray*) – The array to broadcast grouped to.
- **group** (*Union[str, Grouper]*) – Grouping information. See `xclim.sdba.base.Grouper` for details.
- **interp** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method to use,
- **sel** (*Mapping[str, xr.DataArray]*) – Mapping of grouped coordinates to x coordinates (other than the grouping one).

Returns
xr.DataArray

`xclim.sdba.utils.copy_all_attrs(`*ds: xr.Dataset | xr.DataArray, ref: xr.Dataset | xr.DataArray*`)`

Copy all attributes of ds to ref, including attributes of shared coordinates, and variables in the case of Datasets.

`xclim.sdba.utils.ecdf(`*x: DataArray, value: float, dim: str = 'time'*`)` → DataArray

Return the empirical CDF of a sample at a given value.

Parameters

- **x** (*array*) – Sample.
- **value** (*float*) – The value within the support of *x* for which to compute the CDF value.
- **dim** (*str*) – Dimension name.

Returns
xr.DataArray – Empirical CDF.

`xclim.sdba.utils.ensure_longest_doy(`*func: Callable*`)` → Callable

Ensure that selected day is the longest day of year for x and y dims.

`xclim.sdba.utils.equally_spaced_nodes(`*n: int, eps: float | None = None*`)` → np.array

Return nodes with *n* equally spaced points within [0, 1], optionally adding two end-points.

Parameters

- **n** (*int*) – Number of equally spaced nodes.
- **eps** (*float, optional*) – Distance from 0 and 1 of added end nodes. If None (default), do not add endpoints.

Returns

np.array – Nodes between 0 and 1. Nodes can be seen as the middle points of n equal bins.

Warning: Passing a small *eps* will effectively clip the scenario to the bounds of the reference on the historical period in most cases. With normal quantile mapping algorithms, this can give strange result when the reference does not show as many extremes as the simulation does.

Notes

For n=4, eps=0 : 0—x——x——x——x—1

`xclim.sdba.utils.get_clusters(`*data: DataArray*, *u1*, *u2*, *dim: str = 'time'*`)` → Dataset

Get cluster count, maximum and position along a given dim.

See *get_clusters_1d*. Used by *adjustment.ExtremeValues*.

Parameters

- **data** (*1D ndarray*) – Values to get clusters from.
- **u1** (*float*) – Extreme value threshold, at least one value in the cluster must exceed this.
- **u2** (*float*) – Cluster threshold, values above this can be part of a cluster.
- **dim** (*str*) – Dimension name.

Returns

xr.Dataset –

With variables,

- *nclusters* : Number of clusters for each point (with *dim* reduced), int
- *start* : First index in the cluster (*dim* reduced, new *cluster*), int
- *end* : Last index in the cluster, inclusive (*dim* reduced, new *cluster*), int
- *maxpos* : Index of the maximal value within the cluster (*dim* reduced, new *cluster*), int
- *maximum* : Maximal value within the cluster (*dim* reduced, new *cluster*), same dtype as data.

For *start*, *end* and *maxpos*, -1 means NaN and should always correspond to a *NaN* in *maximum*. The length along *cluster* is half the size of "dim", the maximal theoretical number of clusters.

`xclim.sdba.utils.get_clusters_1d(`*data: np.ndarray*, *u1: float*, *u2: float*`)` → tuple[np.array, np.array, np.array, np.array]

Get clusters of a 1D array.

A cluster is defined as a sequence of values larger than u2 with at least one value larger than u1.

Parameters

- **data** (*1D ndarray*) – Values to get clusters from.
- **u1** (*float*) – Extreme value threshold, at least one value in the cluster must exceed this.

- **u2** (*float*) – Cluster threshold, values above this can be part of a cluster.

Returns
(np.array, np.array, np.array, np.array)

References

getcluster of Extremes.jl (read on 2021-04-20) https://github.com/jojal5/Extremes.jl

`xclim.sdba.utils.get_correction(`*x: DataArray*, *y: DataArray*, *kind: str*`)` → DataArray

Return the additive or multiplicative correction/adjustment factors.

`xclim.sdba.utils.interp_on_quantiles(`*newx: xr.DataArray*, *xq: xr.DataArray*, *yq: xr.DataArray*, ***, *group: str | Grouper = 'time'*, *method: str = 'linear'*, *extrapolation: str = 'constant'*`)`

Interpolate values of yq on new values of x.

Interpolate in 2D with `griddata()` if grouping is used, in 1D otherwise, with `interp1d`. Any NaNs in xq or yq are removed from the input map. Similarly, NaNs in newx are left NaNs.

Parameters

- **newx** (*xr.DataArray*) – The values at which to evaluate *yq*. If *group* has group information, *new* should have a coordinate with the same name as the group name In that case, 2D interpolation is used.
- **xq, yq** (*xr.DataArray*) – Coordinates and values on which to interpolate. The interpolation is done along the "quantiles" dimension if *group* has no group information. If it does, interpolation is done in 2D on "quantiles" and on the group dimension.
- **group** (*Union[str, Grouper]*) – The dimension and grouping information. (ex: "time" or "time.month"). Defaults to "time".
- **method** (*{'nearest', 'linear', 'cubic'}*) – The interpolation method.
- **extrapolation** (*{'constant', 'nan'}*) – The extrapolation method used for values of *newx* outside the range of *xq*. See notes.

Notes

Extrapolation methods:

- 'nan' : Any value of *newx* outside the range of *xq* is set to NaN.
- 'constant' : Values of *newx* smaller than the minimum of *xq* are set to the first value of *yq* and those larger than the maximum, set to the last one (first and last non-nan values along the "quantiles" dimension). When the grouping is "time.month", these limits are linearly interpolated along the month dimension.

`xclim.sdba.utils.invert(`*x: xr.DataArray*, *kind: str | None = None*`)` → xr.DataArray

Invert a DataArray either additively (-x) or multiplicatively (1/x).

If kind is not given, default to the one stored in the "kind" attribute of x.

`xclim.sdba.utils.map_cdf(`*ds: Dataset*, ***, *y_value: DataArray*, *dim*`)`

Return the value in *x* with the same CDF as *y_value* in *y*.

This function is meant to be wrapped in a *Grouper.apply*.

Parameters

- **ds** (*xr.Dataset*) – Variables: x, Values from which to pick, y, Reference values giving the ranking
- **y_value** (*float, array*) – Value within the support of *y*.
- **dim** (*str*) – Dimension along which to compute quantile.

Returns
array – Quantile of *x* with the same CDF as *y_value* in *y*.

`xclim.sdba.utils.map_cdf_1d(`*x*, *y*, *y_value*`)`

Return the value in *x* with the same CDF as *y_value* in *y*.

`xclim.sdba.utils.pc_matrix(`*arr: np.ndarray | dsk.Array*`)` → np.ndarray | dsk.Array

Construct a Principal Component matrix.

This matrix can be used to transform points in arr to principal components coordinates. Note that this function does not manage NaNs; if a single observation is null, all elements of the transformation matrix involving that variable will be NaN.

Parameters
arr (*numpy.ndarray or dask.array.Array*) – 2D array (M, N) of the M coordinates of N points.

Returns
numpy.ndarray or dask.array.Array – MxM Array of the same type as arr.

`xclim.sdba.utils.rand_rot_matrix(`*crd: xr.DataArray*, *num: int = 1*, *new_dim: str | None = None*`)` → xr.DataArray

Generate random rotation matrices.

Rotation matrices are members of the SO(n) group, where n is the matrix size (*crd.size*). They can be characterized as orthogonal matrices with determinant 1. A square matrix R is a rotation matrix if and only if $R^t = R^1$ and $\det R = 1$.

Parameters

- **crd** (*xr.DataArray*) – 1D coordinate DataArray along which the rotation occurs. The output will be square with the same coordinate replicated, the second renamed to *new_dim*.
- **num** (*int*) – If larger than 1 (default), the number of matrices to generate, stacked along a "matrices" dimension.
- **new_dim** (*str*) – Name of the new "prime" dimension, defaults to the same name as *crd* + "_prime".

Returns
xr.DataArray – float, NxN if num = 1, numxNxN otherwise, where N is the length of crd.

References

Mezzadri, F. (2006). How to generate random matrices from the classical compact groups. arXiv preprint math-ph/0609050.

`xclim.sdba.utils.rank(`*da: DataArray*, *dim: str = 'time'*, *pct: bool = False*`)` → DataArray

Ranks data along a dimension.

Replicates *xr.DataArray.rank* but as a function usable in a Grouper.apply(). Xarray's docstring is below:

Equal values are assigned a rank that is the average of the ranks that would have been otherwise assigned to all the values within that set. Ranks begin at 1, not 0. If pct, computes percentage ranks.

Parameters

- **da** (*xr.DataArray*) – Source array.
- **dim** (*str, hashable*) – Dimension over which to compute rank.
- **pct** (*bool, optional*) – If True, compute percentage ranks, otherwise compute integer ranks.

Returns

DataArray – DataArray with the same coordinates and dtype 'float64'.

Notes

The *bottleneck* library is required. NaNs in the input array are returned as NaNs.

xclim.testing package

Helpers for testing xclim.

Submodules

xclim.testing.utils module

Testing and tutorial utilities' module.

`xclim.testing.utils.get_all_CMIP6_variables(`*get_cell_methods=True*`)`

`xclim.testing.utils.list_datasets(`*github_repo='Ouranosinc/xclim-testdata'*, *branch='main'*`)`

Return a DataFrame listing all xclim test datasets available on the GitHub repo for the given branch.

The result includes the filepath, as passed to *open_dataset*, the file size (in KB) and the html url to the file. This uses an unauthenticated call to GitHub's REST API, so it is limited to 60 requests per hour (per IP). A single call of this function triggers one request per subdirectory, so use with parsimony.

`xclim.testing.utils.list_input_variables(`*submodules: Optional[Sequence[str]] = None*, *realms: Optional[Sequence[str]] = None*`)` → dict

List all possible variables names used in xclim's indicators.

Made for development purposes. Parses all indicator parameters with the `xclim.core.utils.InputKind.VARIABLE` or *OPTIONAL_VARIABLE* kinds.

Parameters

- **realms** (*Sequence of str, optional*) – Restrict the output to indicators of a list of realms only. Default None, which parses all indicators.
- **submodules** (*str, optional*) – Restrict the output to indicators of a list of submodules only. Default None, which parses all indicators.

Returns
dict – A mapping from variable name to indicator class.

`xclim.testing.utils.open_dataset(`*name: str*, *suffix: str | None = None*, *dap_url: str | None = None*, *github_url: str = 'https://github.com/Ouranosinc/xclim-testdata'*, *branch: str = 'main'*, *cache: bool = True*, *cache_dir: Path = PosixPath('/home/docs/.xclim_testing_data')*, ***kwargs*`)` → Dataset

Open a dataset from the online GitHub-like repository.

If a local copy is found then always use that to avoid network traffic.

Parameters

- **name** (*str*) – Name of the file containing the dataset.
- **suffix** (*str, optional*) – If no suffix is given, assumed to be netCDF ('.nc' is appended). For no suffix, set "".
- **dap_url** (*str, optional*) – URL to OPeNDAP folder where the data is stored. If supplied, supersedes github_url.
- **github_url** (*str*) – URL to GitHub repository where the data is stored.
- **branch** (*str, optional*) – For GitHub-hosted files, the branch to download from.
- **cache_dir** (*Path*) – The directory in which to search for and write cached data.
- **cache** (*bool*) – If True, then cache data locally for use on subsequent calls.
- **kwargs** – For NetCDF files, keywords passed to `xarray.open_dataset()`.

Returns
Union[Dataset, Path]

See also:

`xarray.open_dataset`

`xclim.testing.utils.publish_release_notes(`*style: str = 'md'*, *file: os.PathLike | StringIO | TextIO | None = None*`)` → str | None

Format release history in Markdown or ReStructuredText.

Parameters

- **style** (*{"rst", "md"}*) – Use ReStructuredText formatting or Markdown. Default: Markdown.
- **file** (*{os.PathLike, StringIO, TextIO}, optional*) – If provided, prints to the given file-like object. Otherwise, returns a string.

Returns
str, optional

Notes

This function is solely for development purposes.

`xclim.testing.utils.show_versions`(*file: os.PathLike | StringIO | TextIO | None = None*) → str | None

Print the versions of xclim and its dependencies.

Parameters
file (*{os.PathLike, StringIO, TextIO}, optional*) – If provided, prints to the given file-like object. Otherwise, returns a string.

Returns
str or None

`xclim.testing.utils.update_variable_yaml`(*filename=None*, *xclim_needs_only=True*)

Update a variable from a yaml file.

15.1.2 Submodules

15.1.3 xclim.analog module

Spatial analogues are maps showing which areas have a present-day climate that is analogous to the future climate of a given place. This type of map can be useful for climate adaptation to see how well regions are coping today under specific climate conditions. For example, officials from a city located in a temperate region that may be expecting more heatwaves in the future can learn from the experience of another city where heatwaves are a common occurrence, leading to more proactive intervention plans to better deal with new climate conditions.

Spatial analogues are estimated by comparing the distribution of climate indices computed at the target location over the future period with the distribution of the same climate indices computed over a reference period for multiple candidate regions. A number of methodological choices thus enter the computation:

- Climate indices of interest,
- Metrics measuring the difference between both distributions,
- Reference data from which to compute the base indices,
- A future climate scenario to compute the target indices.

The climate indices chosen to compute the spatial analogues are usually annual values of indices relevant to the intended audience of these maps. For example, in the case of the wine grape industry, the climate indices examined could include the length of the frost-free season, growing degree-days, annual winter minimum temperature and annual number of very cold days [Roy2017].

See *Spatial Analogues examples*.

Methods to compute the (dis)similarity between samples

This module implements all methods described in [Grenier2013] to measure the dissimilarity between two samples, plus the Székely-Rizzo energy distance, Some of these algorithms can be used to test whether two samples have been drawn from the same distribution. Here, they are used in finding areas with analogue climate conditions to a target climate.

- Standardized Euclidean distance
- Nearest Neighbour distance
- Zech-Aslan energy statistic
- Székely-Rizzo energy distance
- Friedman-Rafsky runs statistic
- Kolmogorov-Smirnov statistic
- Kullback-Leibler divergence

All methods accept arrays, the first is the reference (n, D) and the second is the candidate (m, D). Where the climate indicators vary along D and the distribution dimension along n or m. All methods output a single float. See their documentation in *Analogue metrics API*.

Warning: Some methods are scale-invariant and others are not. This is indicated in the docstring of the methods as it can change the results significantly. In most cases, scale-invariance is desirable and inputs may need to be scaled beforehand for scale-dependent methods.

References

`xclim.analog.friedman_rafsky(`*x: ndarray*, *y: ndarray*`)` → float

Compute a dissimilarity metric based on the Friedman-Rafsky runs statistics.

The algorithm builds a minimal spanning tree (the subset of edges connecting all points that minimizes the total edge length) then counts the edges linking points from the same distribution. This method is scale-dependent.

Parameters

- **x** (*np.ndarray (n,d)*) – Reference sample.
- **y** (*np.ndarray (m,d)*) – Candidate sample.

Returns

float – Friedman-Rafsky dissimilarity metric ranging from 0 to (m+n-1)/(m+n).

References

Friedman J.H. and Rafsky, L.C. (1979) Multivariate generalisations of the Wald-Wolfowitz and Smirnov two-sample tests. Annals of Stat. Vol.7, No. 4, 697-717. https://doi.org/10.1214/aos/1176344722.

`xclim.analog.kldiv(`*x: np.ndarray*, *y: np.ndarray*, ***, *k: int | Sequence[int] = 1*`)` → float | Sequence[float]

Compute the Kullback-Leibler divergence between two multivariate samples.

where $r_k(x_i)$ and $s_k(x_i)$ are, respectively, the euclidean distance to the kth neighbour of x_i in the x array (excepting x_i) and in the y array. This method is scale-dependent.

Parameters

- **x** (*np.ndarray (n,d)*) – Samples from distribution P, which typically represents the true distribution (reference).
- **y** (*np.ndarray (m,d)*) – Samples from distribution Q, which typically represents the approximate distribution (candidate)
- **k** (*int or sequence*) – The kth neighbours to look for when estimating the density of the distributions. Defaults to 1, which can be noisy.

Returns
float or sequence – The estimated Kullback-Leibler divergence D(P||Q) computed from the distances to the kth neighbour.

Notes

In information theory, the Kullback–Leibler divergence ([perezcruz08]) is a non-symmetric measure of the difference between two probability distributions P and Q, where P is the "true" distribution and Q an approximation. This nuance is important because $D(P||Q)$ is not equal to $D(Q||P)$.

For probability distributions P and Q of a continuous random variable, the K–L divergence is defined as:

$$D_{KL}(P||Q) = \int p(x) \log \left(\frac{p(x)}{q(x)} \right) dx$$

This formula assumes we have a representation of the probability densities $p(x)$ and $q(x)$. In many cases, we only have samples from the distribution, and most methods first estimate the densities from the samples and then proceed to compute the K-L divergence. In Perez-Cruz, the authors propose an algorithm to estimate the K-L divergence directly from the sample using an empirical CDF. Even though the CDFs do not converge to their true values, the paper proves that the K-L divergence almost surely does converge to its true value.

References

`xclim.analog.kolmogorov_smirnov`(*x: ndarray*, *y: ndarray*) → float

Compute the Kolmogorov-Smirnov statistic applied to two multivariate samples as described by Fasano and Franceschini.

This method is scale-dependent.

Parameters

- **x** (*np.ndarray (n,d)*) – Reference sample.
- **y** (*np.ndarray (m,d)*) – Candidate sample.

Returns
float – Kolmogorov-Smirnov dissimilarity metric ranging from 0 to 1.

References

Fasano, G., & Franceschini, A. (1987). A multidimensional version of the Kolmogorov-Smirnov test. Monthly Notices of the Royal Astronomical Society, 225, 155-170. https://doi.org/10.1093/mnras/225.1.155

`xclim.analog.metric(`*func*`)`

Register a metric function in the *metrics* mapping and add some preparation/checking code.

All metric functions accept 2D inputs. This reshapes 1D inputs to (n, 1) and (m, 1). All metric functions are invalid when any non-finite values are present in the inputs.

`xclim.analog.nearest_neighbor(`*x: ndarray*, *y: ndarray*`)` → ndarray

Compute a dissimilarity metric based on the number of points in the pooled sample whose nearest neighbor belongs to the same distribution.

This method is scale-invariant.

Parameters

- **x** (*np.ndarray (n,d)*) – Reference sample.
- **y** (*np.ndarray (m,d)*) – Candidate sample.

Returns

float – Nearest-Neighbor dissimilarity metric ranging from 0 to 1.

References

Henze N. (1988) A Multivariate two-sample test based on the number of nearest neighbor type coincidences. Ann. of Stat., Vol. 16, No.2, 772-783. https://doi.org/10.1214/aos/1176350835.

`xclim.analog.seuclidean(`*x: ndarray*, *y: ndarray*`)` → float

Compute the Euclidean distance between the mean of a multivariate candidate sample with respect to the mean of a reference sample.

This method is scale-invariant.

Parameters

- **x** (*np.ndarray (n,d)*) – Reference sample.
- **y** (*np.ndarray (m,d)*) – Candidate sample.

Returns

float – Standardized Euclidean Distance between the mean of the samples ranging from 0 to infinity.

Notes

This metric considers neither the information from individual points nor the standard deviation of the candidate distribution.

References

Veloz et al. (2011) Identifying climatic analogs for Wisconsin under 21st-century climate-change scenarios. Climatic Change, https://doi.org/10.1007/s10584-011-0261-z.

`xclim.analog.spatial_analogs(`*target: xr.Dataset*, *candidates: xr.Dataset*, *dist_dim: str | Sequence[str] = 'time'*, *method: str = 'kldiv'*, ***kwargs*`)`

Compute dissimilarity statistics between target points and candidate points.

Spatial analogues based on the comparison of climate indices. The algorithm compares the distribution of the reference indices with the distribution of spatially distributed candidate indices and returns a value measuring the dissimilarity between both distributions over the candidate grid.

Parameters

- **target** (*xr.Dataset*) – Dataset of the target indices. Only indice variables should be included in the dataset's *data_vars*. They should have only the dimension(s) *dist_dim `in common with `candidates*.
- **candidates** (*xr.Dataset*) – Dataset of the candidate indices. Only indice variables should be included in the dataset's *data_vars*.
- **dist_dim** (*str*) – The dimension over which the *distributions* are constructed. This can be a multi-index dimension.
- **method** (*{'seuclidean', 'nearest_neighbor', 'zech_aslan', 'kolmogorov_smirnov', 'friedman_rafsky', 'kldiv'}*) – Which method to use when computing the dissimilarity statistic.
- **kwargs** – Any other parameter passed directly to the dissimilarity method.

Returns

xr.DataArray – The dissimilarity statistic over the union of candidates' and target's dimensions. The range depends on the method.

`xclim.analog.standardize(`*x: np.ndarray*, *y: np.ndarray*`)` → tuple[np.ndarray, np.ndarray]

Standardize x and y by the square root of the product of their standard deviation.

Parameters

- **x** (*np.ndarray*) – Array to be compared.
- **y** (*np.ndarray*) – Array to be compared.

Returns

(ndarray, ndarray) – Standardized arrays.

`xclim.analog.szekely_rizzo(`*x: ndarray*, *y: ndarray*, ***, *standardize: bool = True*`)` → float

Compute the Székely-Rizzo energy distance dissimilarity metric based on an analogy with Newton's gravitational potential energy.

This method is scale-invariant when *standardize=True* (default), scale-dependent otherwise.

Parameters

- **x** (*ndarray (n,d)*) – Reference sample.
- **y** (*ndarray (m,d)*) – Candidate sample.
- **standardize** (*bool*) – If True (default), the standardized euclidean norm is used, instead of the conventional one.

Returns

float – Székely-Rizzo's energy distance dissimilarity metric ranging from 0 to infinity.

Notes

The e-distance between two variables X, Y (target and candidates) of sizes n, d and m, d proposed by [SR2004] is defined by:

$$e(X,Y) = \frac{nm}{n+m}\left[2\phi_{xy}\phi_{xx}\phi_{yy}\right]$$

where

$$\phi_{xy} = \frac{1}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m}\|X_iY_j\|$$
$$\phi_{xx} = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\|X_iX_j\|$$
$$\phi_{yy} = \frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m}\|X_iY_j\|$$

and where $\|\cdot\|$ denotes the Euclidean norm, X_i denotes the i-th observation of X. When *standardized=False*, this corresponds to the T test of [RS2016] (p. 28) and to the `eqdist.e` function of the *energy* R package (with two samples) and gives results twice as big as *xclim.sdba.processing.escore()*. The standardization was added following the logic of [Grenier2013] to make the metric scale-invariant.

References

`xclim.analog.zech_aslan(`*x: ndarray*, *y: ndarray*, ***, *dmin: float = 1e-12*`)` → float

Compute a modified Zech-Aslan energy distance dissimilarity metric based on an analogy with the energy of a cloud of electrical charges.

This method is scale-invariant.

Parameters

- **x** (*np.ndarray (n,d)*) – Reference sample.
- **y** (*np.ndarray (m,d)*) – Candidate sample.
- **dmin** (*float*) – The cut-off for low distances to avoid singularities on identical points.

Returns

float – Zech-Aslan dissimilarity metric ranging from -infinity to infinity.

Notes

The energy measure between two variables X, Y (target and candidates) of sizes n, d and m, d proposed by [AZ03] is defined by:

$$e(X,Y) = \left[\phi_{xx} + \phi_{yy} - \phi_{xy}\right]$$
$$\phi_{xy} = \frac{1}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m}R\left[SED(X_i,Y_j)\right]$$
$$\phi_{xx} = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=i+1}^{n}R\left[SED(X_i,X_j)\right]$$
$$\phi_{yy} = \frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=i+1}^{m}R\left[SED(X_i,Y_j)\right]$$

where X_i denotes the i-th observation of X. R is a weight function and $SED(A, B)$ denotes the standardized Euclidean distance.

$$R(r) = \begin{cases} -\ln r & \text{for } r > d_{min} \\ -\ln d_{min} & \text{for } r \leq d_{min} \end{cases}$$

$$SED(X_i, Y_j) = \sqrt{\sum_{k=1}^{d} \frac{\left(X_i(k) - Y_i(k)\right)^2}{\sigma_x(k)\sigma_y(k)}}$$

where k is a counter over dimensions (indices in the case of spatial analogs) and $\sigma_x(k)$ is the standard deviation of X in dimension k. Finally, d_{min} is a cut-off to avoid poles when $r \to 0$, it is controllable through the *dmin* parameter.

This version corresponds the D_{ZAE} test of [Grenier2013] (eq. 7), which is a version of ϕ_{NM} from [AZ03], modified by using the standardized euclidean distance, the log weight function and choosing $d_{min} = 10^{-12}$.

References

15.1.4 xclim.cli module

xclim command line interface module.

class `xclim.cli.XclimCli`(*name: Optional[str] = None*, *invoke_without_command: bool = False*, *no_args_is_help: Optional[bool] = None*, *subcommand_metavar: Optional[str] = None*, *chain: bool = False*, *result_callback: Optional[Callable[[...], Any]] = None*, ***attrs: Any*)

Bases: `MultiCommand`

Main cli class.

`get_command`(*ctx*, *name*)

Return the requested command.

`list_commands`(*ctx*)

Return the available commands (other than the indicators).

`xclim.cli._create_command`(*indicator_name*)

Generate a Click.Command from an xclim Indicator.

`xclim.cli._format_dict`(*data*, *formatter*, *key_fg='blue'*, *spaces=2*)

`xclim.cli._get_indicator`(*indicator_name*)

`xclim.cli._get_input`(*ctx*)

Return the input dataset stored in the given context.

If the dataset is not open, opens it with open_dataset if a single path was given, or with *open_mfdataset* if a tuple or glob path was given.

`xclim.cli._get_output`(*ctx*)

Return the output dataset stored in the given context.

If the output dataset doesn't exist, create it.

`xclim.cli._process_indicator(`*indicator*, *ctx*, ***params*`)`

Add given climate indicator to the output dataset from variables in the input dataset.

Computation is not triggered here if dask is enabled.

`xclim.cli.write_file(`*ctx*, **args*, ***kwargs*`)`

Write the output dataset to file.

15.1.5 xclim.subset module

Mock subset module for API compatibility.

See also:

`clisops.core.subset`

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[ANUCLIM] https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

[AgbazoGrenier2019] Agbazo, M. N., & Grenier, P. (2019). Characterizing and avoiding physical inconsistency generated by the application of univariate quantile mapping on daily minimum and maximum temperatures over Hudson Bay. International Journal of Climatology, joc.6432. https://doi.org/10.1002/joc.6432

[Grenier2018] Grenier, P. (2018). Two Types of Physical Inconsistency to Avoid with Univariate Quantile Mapping: A Case Study over North America Concerning Relative Humidity and Its Parent Variables. Journal of Applied Meteorology and Climatology, 57(2), 347–364. https://doi.org/10.1175/JAMC-D-17-0177.1

[HoffmanRath2012] Hoffmann, H., & Rath, T. (2012). Meteorologically consistent bias correction of climate time series for agricultural models. Theoretical and Applied Climatology, 110(1–2), 129–141. https://doi.org/10.1007/s00704-012-0618-x

[Thrasher2012] Thrasher, B., Maurer, E. P., McKellar, C., & Duffy, P. B. (2012). Technical Note: Bias correcting climate model simulated daily temperature extremes with quantile mapping. Hydrology and Earth System Sciences, 16(9), 3309–3314. https://doi.org/10.5194/hess-16-3309-2012

[Cannon2015] Cannon, A. J., Sobie, S. R., & Murdock, T. Q. (2015). Bias correction of GCM precipitation by quantile mapping: How well do methods preserve changes in quantiles and extremes? Journal of Climate, 28(17), 6938–6959. https://doi.org/10.1175/JCLI-D-14-00754.1

[RRJF2021] Roy, P., Rondeau-Genesse, G., Jalbert, J., Fournier, É. 2021. Climate Scenarios of Extreme Precipitation Using a Combination of Parametric and Non-Parametric Bias Correction Methods. Submitted to Climate Services, April 2021.

[ClimateTools] https://juliaclimate.github.io/ClimateTools.jl/stable/

[Schmidli2006] Schmidli, J., Frei, C., & Vidale, P. L. (2006). Downscaling from GCM precipitation: A benchmark for dynamical and statistical downscaling methods. International Journal of Climatology, 26(5), 679–689. DOI:10.1002/joc.1287

[Cannon2018] Cannon, A. J. (2018). Multivariate quantile mapping bias correction: An N-dimensional probability density function transform for climate model simulations of multiple variables. Climate Dynamics, 50(1), 31–49. https://doi.org/10.1007/s00382-017-3580-6

[CannonR] https://CRAN.R-project.org/package=MBC

[Mezzadri2006] Mezzadri, F. (2006). How to generate random matrices from the classical compact groups. arXiv preprint math-ph/0609050.

[Pitie2005] Pitie, F., Kokaram, A. C., & Dahyot, R. (2005). N-dimensional probability density function transfer and its application to color transfer. Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1, 2, 1434-1439 Vol. 2. https://doi.org/10.1109/ICCV.2005.166

[SkezelyRizzo2004] Szekely, G. J. and Rizzo, M. L. (2004) Testing for Equal Distributions in High Dimension, InterStat, November (5)

[Hnilica2017] Hnilica, J., Hanel, M. and Ps, V. (2017), Multisite bias correction of precipitation data from regional climate models. Int. J. Climatol., 37: 2934-2946. https://doi.org/10.1002/joc.4890

[Cannon2015] Cannon, A. J., Sobie, S. R., & Murdock, T. Q. (2015). Bias correction of GCM precipitation by quantile mapping: How well do methods preserve changes in quantiles and extremes? Journal of Climate, 28(17), 6938–6959. https://doi.org/10.1175/JCLI-D-14-00754.1

[Themessl2012] Themeßl et al. (2012), Empirical-statistical downscaling and error correction of regional climate models and its impact on the climate change signal, Climatic Change, DOI 10.1007/s10584-011-0224-4.

[BaringhausFranz] Baringhaus, L. and Franz, C. (2004) On a new multivariate two-sample test, Journal of Multivariate Analysis, 88(1), 190–206. https://doi.org/10.1016/s0047-259x(03)00079-4

[Cannon18] Cannon, A. J. (2018). Multivariate quantile mapping bias correction: An N-dimensional probability density function transform for climate model simulations of multiple variables. Climate Dynamics, 50(1), 31–49. https://doi.org/10.1007/s00382-017-3580-6

[SzekelyRizzo] Székely, G. J. and Rizzo, M. L. (2004) Testing for Equal Distributions in High Dimension, InterStat, November (5)

[AlavoineGrenier] Alavoine M., and Grenier P. (under review) The distinct problems of physical inconsistency and of multivariate bias potentially involved in the statistical adjustment of climate simulations. International Journal of Climatology, Manuscript ID: JOC-21-0789, submitted on September 19th 2021. (Preprint https://doi.org/10.31223/X5C34C)

[AlavoineGrenier] Alavoine M., and Grenier P. (under review) The distinct problems of physical inconsistency and of multivariate bias potentially involved in the statistical adjustment of climate simulations. International Journal of Climatology, Manuscript ID: JOC-21-0789, submitted on September 19th 2021. (Preprint https://doi.org/10.31223/X5C34C)

[alavoine2021] Alavoine, M., & Grenier, P. (2021). The distinct problems of physical inconsistency and of multivariate bias potentially involved in the statistical adjustment of climate simulations. https://eartharxiv.org/repository/view/2876/

[hnilica2017] Hnilica, J., Hanel, M. and Ps, V. (2017), Multisite bias correction of precipitation data from regional climate models. Int. J. Climatol., 37: 2934-2946. https://doi.org/10.1002/joc.4890

[Cleveland1979] Cleveland, W. S., 1979. Robust Locally Weighted Regression and Smoothing Scatterplot, Journal of the American Statistical Association 74, 829–836. https://www.doi.org/10.1080/01621459.1979.10481038

[VALUE] http://www.value-cost.eu/

[VALUE] http://www.value-cost.eu/

[Roy2017] Roy, P., Grenier, P., Barriault, E. et al. Climatic Change (2017) 143: 43. https://doi.org/10.1007/s10584-017-1960-x

[Grenier2013] Grenier, P., A.-C. Parent, D. Huard, F. Anctil, and D. Chaumont, 2013: An assessment of six dissimilarity metrics for climate analogs. J. Appl. Meteor. Climatol., 52, 733–752, https://doi.org/10.1175/JAMC-D-12-0170.1

[perezcruz08] Perez-Cruz, F. (2008). Kullback-Leibler divergence estimation of continuous distributions. 2008 IEEE International Symposium on Information Theory, 1666-1670. https://doi.org/10.1109/ISIT.2008.4595271

[SR2004] Székely, G. J. and Rizzo, M. L. (2004) Testing for Equal Distributions in High Dimension, InterStat, November (5). https://www.researchgate.net/publication/228918499_Testing_for_equal_distributions_in_high_dimension

[RS2016] Rizzo, M. L., & Székely, G. J. (2016). Energy distance. Wiley Interdisciplinary Reviews: Computational Statistics, 8(1), 27–38. https://doi.org/10.1002/wics.1375

[AZ03] Aslan B. and Zech G. (2003) A new class of binning-free, multivariate goodness-of-fit tests: the energy tests. https://doi.org/10.48550/arXiv.hep-ex/0203010

[Gladstones1992] Gladstones, J.S. (1992). Viticulture and environment: a study of the effects of environment on grapegrowing and wine qualities, with emphasis on present and future areas for growing winegrapes in Australia. Adelaide: Winetitles.

[ECAD] Project team ECA&D, KNMI (2013). EUMETNET/ECSN optional programme: European Climate Assessment & Dataset (ECA&D) - Algorithm Theoretical Basis Document (ATBD). (KNMI Project number: EPJ029135, v10.7). https://www.ecad.eu/documents/atbd.pdf

[cold_dry_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[cold_wet_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[Blazejczyk2012] Blazejczyk, K., Epstein, Y., Jendritzky, G., Staiger, H., & Tinz, B. (2012). Comparison of UTCI to selected thermal indices. International journal of biometeorology, 56(3), 515-535.

[casati2013] Casati, B., Yagouti, A., & Chaumont, D. (2013). Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. Journal of Applied Meteorology and Climatology, 52(12), 2669-2698. https://doi.org/10.1175/JAMC-D-12-0341.1

[robinson2001] Robinson, P. J. (2001). On the Definition of a Heat Wave. Journal of Applied Meteorology and Climatology, 40(4), 762-775. https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

[casati2013] Casati, B., Yagouti, A., & Chaumont, D. (2013). Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. Journal of Applied Meteorology and Climatology, 52(12), 2669-2698. https://doi.org/10.1175/JAMC-D-12-0341.1

[robinson2001] Robinson, P. J. (2001). On the Definition of a Heat Wave. Journal of Applied Meteorology and Climatology, 40(4), 762-775. https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

[Huglin1978] Huglin, P. (1978). Nouveau mode d'évaluation des possibilités héliothermiques d'un milieu viticole. Dans Symposium International sur l'Écologie de la Vigne (p. 89-98). Ministère de l'Agriculture et de l'Industrie Alimentaire.

[masterton79] Masterton, J. M., & Richardson, F. A. (1979). HUMIDEX, A method of quantifying human discomfort due to excessive heat and humidity, CLI 1-79. Downsview, Ontario: Environment Canada, Atmospheric Environment Service.

[mekis15] Éva Mekis, Lucie A. Vincent, Mark W. Shephard & Xuebin Zhang (2015) Observed Trends in Severe Weather Conditions Based on Humidex, Wind Chill, and Heavy Rainfall Events in Canada for 1953–2012, Atmosphere-Ocean, 53:4, 383-397, DOI: 10.1080/07055900.2015.1086970

[sirangelo20] Sirangelo, B., Caloiero, T., Coscarelli, R. et al. Combining stochastic models of air temperature and vapour pressure for the analysis of the bioclimatic comfort through the Humidex. Sci Rep 10, 11395 (2020). https://doi.org/10.1038/s41598-020-68297-4

[eccc] https://climate.weather.gc.ca/glossary_e.html

[Woollings2010] Woollings, T., Hannachi, A., & Hoskins, B. (2010). Variability of the North Atlantic eddy-driven jet stream. Quarterly Journal of the Royal Meteorological Society, 136(649), 856-868.

[BaierRobertson1965] Baier, W., & Robertson, G. W. (1965). Estimation of latent evaporation from simple weather observations. Canadian journal of plant science, 45(3), 276-284.

[Hargreaves1985] Hargreaves, G. H., & Samani, Z. A. (1985). Reference crop evapotranspiration from temperature. Applied engineering in agriculture, 1(2), 96-99.

[Tanguy2018] Tanguy, M., Prudhomme, C., Smith, K., & Hannaford, J. (2018). Historical gridded reconstruction of potential evapotranspiration for the UK. Earth System Science Data, 10(2), 951-968.

[McGuinness1972] McGuinness, J. L., & Bordne, E. F. (1972). A comparison of lysimeter-derived potential evapotranspiration with computed values (No. 1452). US Department of Agriculture.

[Thornthwaite1948] Thornthwaite, C. W. (1948). An approach toward a rational classification of climate. Geographical review, 38(1), 55-94.

[Lawrence2005] Lawrence, M.G. (2005). The Relationship between Relative Humidity and the Dewpoint Temperature in Moist Air: A Simple Conversion and Applications. Bull. Amer. Meteor. Soc., 86, 225–234, https://doi.org/10.1175/BAMS-86-2-225

[BohrenAlbrecht1998] Craig F. Bohren, Bruce A. Albrecht. Atmospheric Thermodynamics. Oxford University Press, 1998.

[Lawrence2005] Lawrence, M.G. (2005). The Relationship between Relative Humidity and the Dewpoint Temperature in Moist Air: A Simple Conversion and Applications. Bull. Amer. Meteor. Soc., 86, 225–234, https://doi.org/10.1175/BAMS-86-2-225

[BohrenAlbrecht1998] Craig F. Bohren, Bruce A. Albrecht. Atmospheric Thermodynamics. Oxford University Press, 1998.

[goffgratch46] Goff, J. A., and S. Gratch (1946) Low-pressure properties of water from -160 to 212 °F, in Transactions of the American Society of Heating and Ventilating Engineers, pp 95-122, presented at the 52nd annual meeting of the American Society of Heating and Ventilating Engineers, New York, 1946.

[sonntag90] Sonntag, D. (1990). Important new values of the physical constants of 1986, vapour pressure formulations based on the ITS-90, and psychrometer formulae. Zeitschrift für Meteorologie, 40(5), 340-344.

[tetens30] Tetens, O. 1930. Über einige meteorologische Begriffe. Z. Geophys 6: 207-309.

[voemel] https://cires1.colorado.edu/~voemel/vp.html

[wmo08] World Meteorological Organization. (2008). Guide to meteorological instruments and methods of observation. Geneva, Switzerland: World Meteorological Organization. https://www.weather.gov/media/epz/mesonet/CWOP-WMO8.pdf

[its90] Hardy, B. (1998). ITS-90 formulations for vapor pressure, frostpoint temperature, dewpoint temperature, and enhancement factors in the range–100 to+ 100 C. In The Proceedings of the Third International Symposium on Humidity & Moisture (pp. 1-8). https://www.thunderscientific.com/tech_info/reflibrary/its90formulas.pdf

[Verseghy09] Diana Verseghy (2009), CLASS – The Canadian Land Surface Scheme (Version 3.4), Technical Documentation (Version 1.1), Environment Canada, Climate Research Division, Science and Technology Branch.

[WMO08] World Meteorological Organization. (2008). Guide to meteorological instruments and methods of observation. Geneva, Switzerland: World Meteorological Organization. https://www.weather.gov/media/epz/mesonet/CWOP-WMO8.pdf

[WMO08] World Meteorological Organization. (2008). Guide to meteorological instruments and methods of observation. Geneva, Switzerland: World Meteorological Organization. https://www.weather.gov/media/epz/mesonet/CWOP-WMO8.pdf

[warm_dry_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[warm_wet_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[MVSZ2015] Éva Mekis, Lucie A. Vincent, Mark W. Shephard & Xuebin Zhang (2015) Observed Trends in Severe Weather Conditions Based on Humidex, Wind Chill, and Heavy Rainfall Events in Canada for 1953–2012, Atmosphere-Ocean, 53:4, 383-397, DOI: 10.1080/07055900.2015.1086970

[NWS] Wind Chill Questions, Cold Resources, National Weather Service, retrieved 25-05-21. https://www.weather.gov/safety/cold-faqs

[baker2004] Baker, D. B., Richards, R. P., Loftus, T. T., & Kramer, J. W. (2004). A New Flashiness Index: Characteristics and Applications to Midwestern Rivers and Streams1. JAWRA Journal of the American Water Resources Association, 40(2), 503-522. https://doi.org/10.1111/j.1752-1688.2004.tb01046.x

[ANUCLIM] https://fennerschool.anu.edu.au/files/anuclim61.pdf (ch. 6)

[Gladstones1992] Gladstones, J.S. (1992). Viticulture and environment: a study of the effects of environment on grapegrowing and wine qualities, with emphasis on present and future areas for growing winegrapes in Australia. Adelaide: Winetitles.

[ECAD] Project team ECA&D, KNMI (2013). EUMETNET/ECSN optional programme: European Climate Assessment & Dataset (ECA&D) - Algorithm Theoretical Basis Document (ATBD). (KNMI Project number: EPJ029135, v10.7). https://www.ecad.eu/documents/atbd.pdf

[cold_dry_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[cold_wet_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[Blazejczyk2012] Blazejczyk, K., Epstein, Y., Jendritzky, G., Staiger, H., & Tinz, B. (2012). Comparison of UTCI to selected thermal indices. International journal of biometeorology, 56(3), 515-535.

[casati2013] Casati, B., Yagouti, A., & Chaumont, D. (2013). Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. Journal of Applied Meteorology and Climatology, 52(12), 2669-2698. https://doi.org/10.1175/JAMC-D-12-0341.1

[robinson2001] Robinson, P. J. (2001). On the Definition of a Heat Wave. Journal of Applied Meteorology and Climatology, 40(4), 762-775. https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

[casati2013] Casati, B., Yagouti, A., & Chaumont, D. (2013). Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. Journal of Applied Meteorology and Climatology, 52(12), 2669-2698. https://doi.org/10.1175/JAMC-D-12-0341.1

[robinson2001] Robinson, P. J. (2001). On the Definition of a Heat Wave. Journal of Applied Meteorology and Climatology, 40(4), 762-775. https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

[Huglin1978] Huglin, P. (1978). Nouveau mode d'évaluation des possibilités héliothermiques d'un milieu viticole. Dans Symposium International sur l'Écologie de la Vigne (p. 89-98). Ministère de l'Agriculture et de l'Industrie Alimentaire.

[masterton79] Masterton, J. M., & Richardson, F. A. (1979). HUMIDEX, A method of quantifying human discomfort due to excessive heat and humidity, CLI 1-79. Downsview, Ontario: Environment Canada, Atmospheric Environment Service.

[mekis15] Éva Mekis, Lucie A. Vincent, Mark W. Shephard & Xuebin Zhang (2015) Observed Trends in Severe Weather Conditions Based on Humidex, Wind Chill, and Heavy Rainfall Events in Canada for 1953–2012, Atmosphere-Ocean, 53:4, 383-397, DOI: 10.1080/07055900.2015.1086970

[sirangelo20] Sirangelo, B., Caloiero, T., Coscarelli, R. et al. Combining stochastic models of air temperature and vapour pressure for the analysis of the bioclimatic comfort through the Humidex. Sci Rep 10, 11395 (2020). https://doi.org/10.1038/s41598-020-68297-4

[eccc] https://climate.weather.gc.ca/glossary_e.html

[Woollings2010] Woollings, T., Hannachi, A., & Hoskins, B. (2010). Variability of the North Atlantic eddy-driven jet stream. Quarterly Journal of the Royal Meteorological Society, 136(649), 856-868.

[BaierRobertson1965] Baier, W., & Robertson, G. W. (1965). Estimation of latent evaporation from simple weather observations. Canadian journal of plant science, 45(3), 276-284.

[Hargreaves1985] Hargreaves, G. H., & Samani, Z. A. (1985). Reference crop evapotranspiration from temperature. Applied engineering in agriculture, 1(2), 96-99.

[Tanguy2018] Tanguy, M., Prudhomme, C., Smith, K., & Hannaford, J. (2018). Historical gridded reconstruction of potential evapotranspiration for the UK. Earth System Science Data, 10(2), 951-968.

[McGuinness1972] McGuinness, J. L., & Bordne, E. F. (1972). A comparison of lysimeter-derived potential evapotranspiration with computed values (No. 1452). US Department of Agriculture.

[Thornthwaite1948] Thornthwaite, C. W. (1948). An approach toward a rational classification of climate. Geographical review, 38(1), 55-94.

[baker2004] Baker, D. B., Richards, R. P., Loftus, T. T., & Kramer, J. W. (2004). A New Flashiness Index: Characteristics and Applications to Midwestern Rivers and Streams1. JAWRA Journal of the American Water Resources Association, 40(2), 503-522. https://doi.org/10.1111/j.1752-1688.2004.tb01046.x

[Lawrence2005] Lawrence, M.G. (2005). The Relationship between Relative Humidity and the Dewpoint Temperature in Moist Air: A Simple Conversion and Applications. Bull. Amer. Meteor. Soc., 86, 225–234, https://doi.org/10.1175/BAMS-86-2-225

[BohrenAlbrecht1998] Craig F. Bohren, Bruce A. Albrecht. Atmospheric Thermodynamics. Oxford University Press, 1998.

[goffgratch46] Goff, J. A., and S. Gratch (1946) Low-pressure properties of water from -160 to 212 °F, in Transactions of the American Society of Heating and Ventilating Engineers, pp 95-122, presented at the 52nd annual meeting of the American Society of Heating and Ventilating Engineers, New York, 1946.

[sonntag90] Sonntag, D. (1990). Important new values of the physical constants of 1986, vapour pressure formulations based on the ITS-90, and psychrometer formulae. Zeitschrift für Meteorologie, 40(5), 340-344.

[tetens30] Tetens, O. 1930. Über einige meteorologische Begriffe. Z. Geophys 6: 207-309.

[voemel] https://cires1.colorado.edu/~voemel/vp.html

[wmo08] World Meteorological Organization. (2008). Guide to meteorological instruments and methods of observation. Geneva, Switzerland: World Meteorological Organization. https://www.weather.gov/media/epz/mesonet/CWOP-WMO8.pdf

[its90] Hardy, B. (1998). ITS-90 formulations for vapor pressure, frostpoint temperature, dewpoint temperature, and enhancement factors in the range–100 to+ 100 C. In The Proceedings of the Third International Symposium on Humidity & Moisture (pp. 1-8). https://www.thunderscientific.com/tech_info/reflibrary/its90formulas.pdf

[Verseghy09] Diana Verseghy (2009), CLASS – The Canadian Land Surface Scheme (Version 3.4), Technical Documentation (Version 1.1), Environment Canada, Climate Research Division, Science and Technology Branch.

[WMO08] World Meteorological Organization. (2008). Guide to meteorological instruments and methods of observation. Geneva, Switzerland: World Meteorological Organization. https://www.weather.gov/media/epz/mesonet/CWOP-WMO8.pdf

[WMO08] World Meteorological Organization. (2008). Guide to meteorological instruments and methods of observation. Geneva, Switzerland: World Meteorological Organization. https://www.weather.gov/media/epz/mesonet/CWOP-WMO8.pdf

[warm_dry_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[warm_wet_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[MVSZ2015] Éva Mekis, Lucie A. Vincent, Mark W. Shephard & Xuebin Zhang (2015) Observed Trends in Severe Weather Conditions Based on Humidex, Wind Chill, and Heavy Rainfall Events in Canada for 1953–2012, Atmosphere-Ocean, 53:4, 383-397, DOI: 10.1080/07055900.2015.1086970

[NWS] Wind Chill Questions, Cold Resources, National Weather Service, retrieved 25-05-21. https://www.weather.gov/safety/cold-faqs

[cffdrs] Cantin, A., Wang, X., Parisien M-A., Wotton, M., Anderson, K., Moore, B., Schiks, T., Flannigan, M., Canadian Forest Fire Danger Rating System, R package, CRAN, https://cran.r-project.org/package=cffdrs

[GFWED2015] Field, R. D., Spessa, A. C., Aziz, N. A., Camia, A., Cantin, A., Carr, R., de Groot, W. J., Dowdy, A. J., Flannigan, M. D., Manomaiphiboon, K., Pappenberger, F., Tanpipat, V., and Wang, X. (2015) Development of a Global Fire Weather Database, Nat. Hazards Earth Syst. Sci., 15, 1407–1423, https://doi.org/10.5194/nhess-15-1407-2015

[VanWagner1985] Van Wagner, C.E. 1985. Drought, timelag and fire danger rating. Pages 178-185 in L.R. Donoghueand R.E. Martin, eds. Proc. 8th Conf. Fire For. Meteorol., 29 Apr.-3 May 1985, Detroit, MI. Soc.Am. For., Bethesda, MD.http://cfs.nrcan.gc.ca/pubwarehouse/pdfs/23550.pdf

[MBHFJ2020] McElhinny, M., Beckers, J. F., Hanes, C., Flannigan, M., and Jain, P. 2020. A high-resolution reanalysis of global fire weather from 1979 to 2018 – overwintering the Drought Code, Earth Syst. Sci. Data, 12, 1823–1833, https://doi.org/10.5194/essd-12-1823-2020

[Matthes17] Matthes, K. et al. (2017). Solar forcing for CMIP6 (v3.2). Geoscientific Model Development, 10(6), 2247–2302. https://doi.org/10.5194/gmd-10-2247-2017

[Kalogirou14] Kalogirou, S. A. (2014). Chapter 2 — Environmental Characteristics. In S. A. Kalogirou (Ed.), Solar Energy Engineering (Second Edition) (pp. 51–123). Academic Press. https://doi.org/10.1016/B978-0-12-397270-5.00002-9

[Spencer1971] Spencer JW (1971) Fourier series representation of the position of the sun. Search 2(5):172

[CannonKKZ] Cannon, Alex J. (2015). Selecting GCM Scenarios that Span the Range of Changes in a Multimodel Ensemble: Application to CMIP5 Climate Extremes Indices. Journal of Climate, (28)3, 1260-1267. https://doi.org/10.1175/JCLI-D-14-00636.1

[AR5WG1C12] https://www.ipcc.ch/site/assets/uploads/2018/02/WG1AR5_Chapter12_FINAL.pdf

[tebaldi2011] Tebaldi C., Arblaster, J.M. and Knutti, R. (2011) Mapping model agreement on future climate projections. GRL. doi:10.1029/2011GL049863

[knutti2013] Knutti, R. and Sedlácek, J. (2013) Robustness and uncertainties in the new CMIP5 climate model projections. Nat. Clim. Change. doi:10.1038/nclimate1716

[LOCA] Pierce, D. W., D. R. Cayan, and B. L. Thrasher, 2014: Statistical downscaling using Localized Constructed Analogs (LOCA). Journal of Hydrometeorology, volume 15, page 2558-2585

[HyndmanFan] Hyndman, R. J., & Fan, Y. (1996). Sample quantiles in statistical packages. The American Statistician, 50(4), 361-365.

[LOCA] Pierce, D. W., D. R. Cayan, and B. L. Thrasher, 2014: Statistical downscaling using Localized Constructed Analogs (LOCA). Journal of Hydrometeorology, volume 15, page 2558-2585

[HyndmanFan] Hyndman, R. J., & Fan, Y. (1996). Sample quantiles in statistical packages. The American Statistician, 50(4), 361-365.

[CannonKKZ] Cannon, Alex J. (2015). Selecting GCM Scenarios that Span the Range of Changes in a Multimodel Ensemble: Application to CMIP5 Climate Extremes Indices. Journal of Climate, (28)3, 1260-1267. https://doi.org/10.1175/JCLI-D-14-00636.1

[AR5WG1C12] https://www.ipcc.ch/site/assets/uploads/2018/02/WG1AR5_Chapter12_FINAL.pdf

[tebaldi2011] Tebaldi C., Arblaster, J.M. and Knutti, R. (2011) Mapping model agreement on future climate projections. GRL. doi:10.1029/2011GL049863

[knutti2013] Knutti, R. and Sedlácek, J. (2013) Robustness and uncertainties in the new CMIP5 climate model projections. Nat. Clim. Change. doi:10.1038/nclimate1716

[Blazejczyk2012] Blazejczyk, K., Epstein, Y., Jendritzky, G., Staiger, H., & Tinz, B. (2012). Comparison of UTCI to selected thermal indices. International journal of biometeorology, 56(3), 515-535.

[masterton79] Masterton, J. M., & Richardson, F. A. (1979). HUMIDEX, A method of quantifying human discomfort due to excessive heat and humidity, CLI 1-79. Downsview, Ontario: Environment Canada, Atmospheric Environment Service.

[mekis15] Éva Mekis, Lucie A. Vincent, Mark W. Shephard & Xuebin Zhang (2015) Observed Trends in Severe Weather Conditions Based on Humidex, Wind Chill, and Heavy Rainfall Events in Canada for 1953–2012, Atmosphere-Ocean, 53:4, 383-397, DOI: 10.1080/07055900.2015.1086970

[sirangelo20] Sirangelo, B., Caloiero, T., Coscarelli, R. et al. Combining stochastic models of air temperature and vapour pressure for the analysis of the bioclimatic comfort through the Humidex. Sci Rep 10, 11395 (2020). https://doi.org/10.1038/s41598-020-68297-4

[eccc] https://climate.weather.gc.ca/glossary_e.html

[BaierRobertson1965] Baier, W., & Robertson, G. W. (1965). Estimation of latent evaporation from simple weather observations. Canadian journal of plant science, 45(3), 276-284.

[Hargreaves1985] Hargreaves, G. H., & Samani, Z. A. (1985). Reference crop evapotranspiration from temperature. Applied engineering in agriculture, 1(2), 96-99.

[Tanguy2018] Tanguy, M., Prudhomme, C., Smith, K., & Hannaford, J. (2018). Historical gridded reconstruction of potential evapotranspiration for the UK. Earth System Science Data, 10(2), 951-968.

[McGuinness1972] McGuinness, J. L., & Bordne, E. F. (1972). A comparison of lysimeter-derived potential evapotranspiration with computed values (No. 1452). US Department of Agriculture.

[Thornthwaite1948] Thornthwaite, C. W. (1948). An approach toward a rational classification of climate. Geographical review, 38(1), 55-94.

[Lawrence2005] Lawrence, M.G. (2005). The Relationship between Relative Humidity and the Dewpoint Temperature in Moist Air: A Simple Conversion and Applications. Bull. Amer. Meteor. Soc., 86, 225–234, https://doi.org/10.1175/BAMS-86-2-225

[BohrenAlbrecht1998] Craig F. Bohren, Bruce A. Albrecht. Atmospheric Thermodynamics. Oxford University Press, 1998.

[Lawrence2005] Lawrence, M.G. (2005). The Relationship between Relative Humidity and the Dewpoint Temperature in Moist Air: A Simple Conversion and Applications. Bull. Amer. Meteor. Soc., 86, 225–234, https://doi.org/10.1175/BAMS-86-2-225

[BohrenAlbrecht1998] Craig F. Bohren, Bruce A. Albrecht. Atmospheric Thermodynamics. Oxford University Press, 1998.

[goffgratch46] Goff, J. A., and S. Gratch (1946) Low-pressure properties of water from -160 to 212 °F, in Transactions of the American Society of Heating and Ventilating Engineers, pp 95-122, presented at the 52nd annual meeting of the American Society of Heating and Ventilating Engineers, New York, 1946.

[sonntag90] Sonntag, D. (1990). Important new values of the physical constants of 1986, vapour pressure formulations based on the ITS-90, and psychrometer formulae. Zeitschrift für Meteorologie, 40(5), 340-344.

[tetens30] Tetens, O. 1930. Über einige meteorologische Begriffe. Z. Geophys 6: 207-309.

[voemel] https://cires1.colorado.edu/~voemel/vp.html

[wmo08] World Meteorological Organization. (2008). Guide to meteorological instruments and methods of observation. Geneva, Switzerland: World Meteorological Organization. https://www.weather.gov/media/epz/mesonet/CWOP-WMO8.pdf

[its90] Hardy, B. (1998). ITS-90 formulations for vapor pressure, frostpoint temperature, dewpoint temperature, and enhancement factors in the range–100 to+ 100 C. In The Proceedings of the Third International Symposium on Humidity & Moisture (pp. 1-8). https://www.thunderscientific.com/tech_info/reflibrary/its90formulas.pdf

[Verseghy09] Diana Verseghy (2009), CLASS – The Canadian Land Surface Scheme (Version 3.4), Technical Documentation (Version 1.1), Environment Canada, Climate Research Division, Science and Technology Branch.

[WMO08] World Meteorological Organization. (2008). Guide to meteorological instruments and methods of observation. Geneva, Switzerland: World Meteorological Organization. https://www.weather.gov/media/epz/mesonet/CWOP-WMO8.pdf

[WMO08] World Meteorological Organization. (2008). Guide to meteorological instruments and methods of observation. Geneva, Switzerland: World Meteorological Organization. https://www.weather.gov/media/epz/mesonet/CWOP-WMO8.pdf

[MVSZ2015] Éva Mekis, Lucie A. Vincent, Mark W. Shephard & Xuebin Zhang (2015) Observed Trends in Severe Weather Conditions Based on Humidex, Wind Chill, and Heavy Rainfall Events in Canada for 1953–2012, Atmosphere-Ocean, 53:4, 383-397, DOI: 10.1080/07055900.2015.1086970

[NWS] Wind Chill Questions, Cold Resources, National Weather Service, retrieved 25-05-21. https://www.weather.gov/safety/cold-faqs

[cold_dry_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119 [cold_wet_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[warm_dry_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[warm_wet_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[Woollings2010] Woollings, T., Hannachi, A., & Hoskins, B. (2010). Variability of the North Atlantic eddy-driven jet stream. Quarterly Journal of the Royal Meteorological Society, 136(649), 856-868.

[Gladstones1992] Gladstones, J.S. (1992). Viticulture and environment: a study of the effects of environment on grapegrowing and wine qualities, with emphasis on present and future areas for growing winegrapes in Australia. Adelaide: Winetitles.

[ECAD] Project team ECA&D, KNMI (2013). EUMETNET/ECSN optional programme: European Climate Assessment & Dataset (ECA&D) - Algorithm Theoretical Basis Document (ATBD). (KNMI Project number: EPJ029135, v10.7). https://www.ecad.eu/documents/atbd.pdf

[casati2013] Casati, B., Yagouti, A., & Chaumont, D. (2013). Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. Journal of Applied Meteorology and Climatology, 52(12), 2669-2698. https://doi.org/10.1175/JAMC-D-12-0341.1

[robinson2001] Robinson, P. J. (2001). On the Definition of a Heat Wave. Journal of Applied Meteorology and Climatology, 40(4), 762-775. https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

[casati2013] Casati, B., Yagouti, A., & Chaumont, D. (2013). Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. Journal of Applied Meteorology and Climatology, 52(12), 2669-2698. https://doi.org/10.1175/JAMC-D-12-0341.1

[robinson2001] Robinson, P. J. (2001). On the Definition of a Heat Wave. Journal of Applied Meteorology and Climatology, 40(4), 762-775. https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

[Huglin1978] Huglin, P. (1978). Nouveau mode d'évaluation des possibilités héliothermiques d'un milieu viticole. Dans Symposium International sur l'Écologie de la Vigne (p. 89-98). Ministère de l'Agriculture et de l'Industrie Alimentaire.

[baker2004] Baker, D. B., Richards, R. P., Loftus, T. T., & Kramer, J. W. (2004). A New Flashiness Index: Characteristics and Applications to Midwestern Rivers and Streams1. JAWRA Journal of the American Water Resources Association, 40(2), 503-522. https://doi.org/10.1111/j.1752-1688.2004.tb01046.x

[Gladstones1992] Gladstones, J.S. (1992). Viticulture and environment: a study of the effects of environment on grapegrowing and wine qualities, with emphasis on present and future areas for growing winegrapes in Australia. Adelaide: Winetitles.

[ECAD] Project team ECA&D, KNMI (2013). EUMETNET/ECSN optional programme: European Climate Assessment & Dataset (ECA&D) - Algorithm Theoretical Basis Document (ATBD). (KNMI Project number: EPJ029135, v10.7). https://www.ecad.eu/documents/atbd.pdf

[Huglin1978] Huglin, P. (1978). Nouveau mode d'évaluation des possibilités héliothermiques d'un milieu viticole. Dans Symposium International sur l'Écologie de la Vigne (p. 89-98). Ministère de l'Agriculture et de l'Industrie Alimentaire.

[Blazejczyk2012] Blazejczyk, K., Epstein, Y., Jendritzky, G., Staiger, H., & Tinz, B. (2012). Comparison of UTCI to selected thermal indices. International journal of biometeorology, 56(3), 515-535.

[masterton79] Masterton, J. M., & Richardson, F. A. (1979). HUMIDEX, A method of quantifying human discomfort due to excessive heat and humidity, CLI 1-79. Downsview, Ontario: Environment Canada, Atmospheric Environment Service.

[mekis15] Éva Mekis, Lucie A. Vincent, Mark W. Shephard & Xuebin Zhang (2015) Observed Trends in Severe Weather Conditions Based on Humidex, Wind Chill, and Heavy Rainfall Events in Canada for 1953–2012, Atmosphere-Ocean, 53:4, 383-397, DOI: 10.1080/07055900.2015.1086970

[sirangelo20] Sirangelo, B., Caloiero, T., Coscarelli, R. et al. Combining stochastic models of air temperature and vapour pressure for the analysis of the bioclimatic comfort through the Humidex. Sci Rep 10, 11395 (2020). https://doi.org/10.1038/s41598-020-68297-4

[eccc] https://climate.weather.gc.ca/glossary_e.html

[BaierRobertson1965] Baier, W., & Robertson, G. W. (1965). Estimation of latent evaporation from simple weather observations. Canadian journal of plant science, 45(3), 276-284.

[Hargreaves1985] Hargreaves, G. H., & Samani, Z. A. (1985). Reference crop evapotranspiration from temperature. Applied engineering in agriculture, 1(2), 96-99.

[Tanguy2018] Tanguy, M., Prudhomme, C., Smith, K., & Hannaford, J. (2018). Historical gridded reconstruction of potential evapotranspiration for the UK. Earth System Science Data, 10(2), 951-968.

[McGuinness1972] McGuinness, J. L., & Bordne, E. F. (1972). A comparison of lysimeter-derived potential evapotranspiration with computed values (No. 1452). US Department of Agriculture.

[Thornthwaite1948] Thornthwaite, C. W. (1948). An approach toward a rational classification of climate. Geographical review, 38(1), 55-94.

[Lawrence2005] Lawrence, M.G. (2005). The Relationship between Relative Humidity and the Dewpoint Temperature in Moist Air: A Simple Conversion and Applications. Bull. Amer. Meteor. Soc., 86, 225–234, https://doi.org/10.1175/BAMS-86-2-225

[BohrenAlbrecht1998] Craig F. Bohren, Bruce A. Albrecht. Atmospheric Thermodynamics. Oxford University Press, 1998.

[goffgratch46] Goff, J. A., and S. Gratch (1946) Low-pressure properties of water from -160 to 212 °F, in Transactions of the American Society of Heating and Ventilating Engineers, pp 95-122, presented at the 52nd annual meeting of the American Society of Heating and Ventilating Engineers, New York, 1946.

[sonntag90] Sonntag, D. (1990). Important new values of the physical constants of 1986, vapour pressure formulations based on the ITS-90, and psychrometer formulae. Zeitschrift für Meteorologie, 40(5), 340-344.

[tetens30] Tetens, O. 1930. Über einige meteorologische Begriffe. Z. Geophys 6: 207-309.

[voemel] https://cires1.colorado.edu/~voemel/vp.html

[wmo08] World Meteorological Organization. (2008). Guide to meteorological instruments and methods of observation. Geneva, Switzerland: World Meteorological Organization. https://www.weather.gov/media/epz/mesonet/CWOP-WMO8.pdf

[its90] Hardy, B. (1998). ITS-90 formulations for vapor pressure, frostpoint temperature, dewpoint temperature, and enhancement factors in the range–100 to+ 100 C. In The Proceedings of the Third International Symposium on Humidity & Moisture (pp. 1-8). https://www.thunderscientific.com/tech_info/reflibrary/its90formulas.pdf

[Verseghy09] Diana Verseghy (2009), CLASS – The Canadian Land Surface Scheme (Version 3.4), Technical Documentation (Version 1.1), Environment Canada, Climate Research Division, Science and Technology Branch.

[WMO08] World Meteorological Organization. (2008). Guide to meteorological instruments and methods of observation. Geneva, Switzerland: World Meteorological Organization. https://www.weather.gov/media/epz/mesonet/CWOP-WMO8.pdf

[WMO08] World Meteorological Organization. (2008). Guide to meteorological instruments and methods of observation. Geneva, Switzerland: World Meteorological Organization. https://www.weather.gov/media/epz/mesonet/CWOP-WMO8.pdf

[MVSZ2015] Éva Mekis, Lucie A. Vincent, Mark W. Shephard & Xuebin Zhang (2015) Observed Trends in Severe Weather Conditions Based on Humidex, Wind Chill, and Heavy Rainfall Events in Canada for 1953–2012, Atmosphere-Ocean, 53:4, 383-397, DOI: 10.1080/07055900.2015.1086970

[NWS] Wind Chill Questions, Cold Resources, National Weather Service, retrieved 25-05-21. https://www.weather.gov/safety/cold-faqs

[baker2004] Baker, D. B., Richards, R. P., Loftus, T. T., & Kramer, J. W. (2004). A New Flashiness Index: Characteristics and Applications to Midwestern Rivers and Streams1. JAWRA Journal of the American Water Resources Association, 40(2), 503-522. https://doi.org/10.1111/j.1752-1688.2004.tb01046.x

[cold_dry_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[cold_wet_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[casati2013] Casati, B., Yagouti, A., & Chaumont, D. (2013). Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. Journal of Applied Meteorology and Climatology, 52(12), 2669-2698. https://doi.org/10.1175/JAMC-D-12-0341.1

[robinson2001] Robinson, P. J. (2001). On the Definition of a Heat Wave. Journal of Applied Meteorology and Climatology, 40(4), 762-775. https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

[casati2013] Casati, B., Yagouti, A., & Chaumont, D. (2013). Regional Climate Projections of Extreme Heat Events in Nine Pilot Canadian Communities for Public Health Planning. Journal of Applied Meteorology and Climatology, 52(12), 2669-2698. https://doi.org/10.1175/JAMC-D-12-0341.1

[robinson2001] Robinson, P. J. (2001). On the Definition of a Heat Wave. Journal of Applied Meteorology and Climatology, 40(4), 762-775. https://doi.org/10.1175/1520-0450%282001%29040<0762:OTDOAH>2.0.CO;2

[warm_dry_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[warm_wet_days] Beniston, M. (2009). Trends in joint quantiles of temperature and precipitation in Europe since 1901 and projected for 2100. Geophysical Research Letters, 36(7). https://doi.org/10.1029/2008GL037119

[Woollings2010] Woollings, T., Hannachi, A., & Hoskins, B. (2010). Variability of the North Atlantic eddy-driven jet stream. Quarterly Journal of the Royal Meteorological Society, 136(649), 856-868.

[cffdrs] Cantin, A., Wang, X., Parisien M-A., Wotton, M., Anderson, K., Moore, B., Schiks, T., Flannigan, M., Canadian Forest Fire Danger Rating System, R package, CRAN, https://cran.r-project.org/package=cffdrs

[GFWED2015] Field, R. D., Spessa, A. C., Aziz, N. A., Camia, A., Cantin, A., Carr, R., de Groot, W. J., Dowdy, A. J., Flannigan, M. D., Manomaiphiboon, K., Pappenberger, F., Tanpipat, V., and Wang, X. (2015) Development of a Global Fire Weather Database, Nat. Hazards Earth Syst. Sci., 15, 1407–1423, https://doi.org/10.5194/nhess-15-1407-2015

[VanWagner1985] Van Wagner, C.E. 1985. Drought, timelag and fire danger rating. Pages 178-185 in L.R. Donoghueand R.E. Martin, eds. Proc. 8th Conf. Fire For. Meteorol., 29 Apr.-3 May 1985, Detroit, MI. Soc.Am. For., Bethesda, MD.http://cfs.nrcan.gc.ca/pubwarehouse/pdfs/23550.pdf

[MBHFJ2020] McElhinny, M., Beckers, J. F., Hanes, C., Flannigan, M., and Jain, P. 2020. A high-resolution reanalysis of global fire weather from 1979 to 2018 – overwintering the Drought Code, Earth Syst. Sci. Data, 12, 1823–1833, https://doi.org/10.5194/essd-12-1823-2020

[Matthes17] Matthes, K. et al. (2017). Solar forcing for CMIP6 (v3.2). Geoscientific Model Development, 10(6), 2247–2302. https://doi.org/10.5194/gmd-10-2247-2017

[Kalogirou14] Kalogirou, S. A. (2014). Chapter 2 — Environmental Characteristics. In S. A. Kalogirou (Ed.), Solar Energy Engineering (Second Edition) (pp. 51–123). Academic Press. https://doi.org/10.1016/B978-0-12-397270-5.00002-9

[Spencer1971] Spencer JW (1971) Fourier series representation of the position of the sun. Search 2(5):172

[Cannon2015] Cannon, A. J., Sobie, S. R., & Murdock, T. Q. (2015). Bias correction of GCM precipitation by quantile mapping: How well do methods preserve changes in quantiles and extremes? Journal of Climate, 28(17), 6938–6959. https://doi.org/10.1175/JCLI-D-14-00754.1

[RRJF2021] Roy, P., Rondeau-Genesse, G., Jalbert, J., Fournier, É. 2021. Climate Scenarios of Extreme Precipitation Using a Combination of Parametric and Non-Parametric Bias Correction Methods. Submitted to Climate Services, April 2021.

[ClimateTools] https://juliaclimate.github.io/ClimateTools.jl/stable/

[Schmidli2006] Schmidli, J., Frei, C., & Vidale, P. L. (2006). Downscaling from GCM precipitation: A benchmark for dynamical and statistical downscaling methods. International Journal of Climatology, 26(5), 679–689. DOI:10.1002/joc.1287

[Cannon2018] Cannon, A. J. (2018). Multivariate quantile mapping bias correction: An N-dimensional probability density function transform for climate model simulations of multiple variables. Climate Dynamics, 50(1), 31–49. https://doi.org/10.1007/s00382-017-3580-6

[CannonR] https://CRAN.R-project.org/package=MBC

[Mezzadri2006] Mezzadri, F. (2006). How to generate random matrices from the classical compact groups. arXiv preprint math-ph/0609050.

[Pitie2005] Pitie, F., Kokaram, A. C., & Dahyot, R. (2005). N-dimensional probability density function transfer and its application to color transfer. Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1, 2, 1434-1439 Vol. 2. https://doi.org/10.1109/ICCV.2005.166

[SkezelyRizzo2004] Szekely, G. J. and Rizzo, M. L. (2004) Testing for Equal Distributions in High Dimension, InterStat, November (5)

[Hnilica2017] Hnilica, J., Hanel, M. and Ps, V. (2017), Multisite bias correction of precipitation data from regional climate models. Int. J. Climatol., 37: 2934-2946. https://doi.org/10.1002/joc.4890

[Cannon2015] Cannon, A. J., Sobie, S. R., & Murdock, T. Q. (2015). Bias correction of GCM precipitation by quantile mapping: How well do methods preserve changes in quantiles and extremes? Journal of Climate, 28(17), 6938–6959. https://doi.org/10.1175/JCLI-D-14-00754.1

[Cleveland1979] Cleveland, W. S., 1979. Robust Locally Weighted Regression and Smoothing Scatterplot, Journal of the American Statistical Association 74, 829–836. https://www.doi.org/10.1080/01621459.1979.10481038.

[Cleveland1979] Cleveland, W. S., 1979. Robust Locally Weighted Regression and Smoothing Scatterplot, Journal of the American Statistical Association 74, 829–836. https://www.doi.org/10.1080/01621459.1979.10481038

[VALUE] http://www.value-cost.eu/

[Themessl2012] Themeßl et al. (2012), Empirical-statistical downscaling and error correction of regional climate models and its impact on the climate change signal, Climatic Change, DOI 10.1007/s10584-011-0224-4.

[BaringhausFranz] Baringhaus, L. and Franz, C. (2004) On a new multivariate two-sample test, Journal of Multivariate Analysis, 88(1), 190–206. https://doi.org/10.1016/s0047-259x(03)00079-4

[Cannon18] Cannon, A. J. (2018). Multivariate quantile mapping bias correction: An N-dimensional probability density function transform for climate model simulations of multiple variables. Climate Dynamics, 50(1), 31–49. https://doi.org/10.1007/s00382-017-3580-6

[SzekelyRizzo] Székely, G. J. and Rizzo, M. L. (2004) Testing for Equal Distributions in High Dimension, InterStat, November (5)

[AlavoineGrenier] Alavoine M., and Grenier P. (under review) The distinct problems of physical inconsistency and of multivariate bias potentially involved in the statistical adjustment of climate simulations. International Journal of Climatology, Manuscript ID: JOC-21-0789, submitted on September 19th 2021. (Preprint https://doi.org/10.31223/X5C34C)

[AlavoineGrenier] Alavoine M., and Grenier P. (under review) The distinct problems of physical inconsistency and of multivariate bias potentially involved in the statistical adjustment of climate simulations. International Journal of Climatology, Manuscript ID: JOC-21-0789, submitted on September 19th 2021. (Preprint https://doi.org/10.31223/X5C34C)

[VALUE] http://www.value-cost.eu/

[alavoine2021] Alavoine, M., & Grenier, P. (2021). The distinct problems of physical inconsistency and of multivariate bias potentially involved in the statistical adjustment of climate simulations. https://eartharxiv.org/repository/view/2876/

[hnilica2017] Hnilica, J., Hanel, M. and Ps, V. (2017), Multisite bias correction of precipitation data from regional climate models. Int. J. Climatol., 37: 2934-2946. https://doi.org/10.1002/joc.4890

[Roy2017] Roy, P., Grenier, P., Barriault, E. et al. Climatic Change (2017) 143: 43. https://doi.org/10.1007/s10584-017-1960-x

[Grenier2013] Grenier, P., A.-C. Parent, D. Huard, F. Anctil, and D. Chaumont, 2013: An assessment of six dissimilarity metrics for climate analogs. J. Appl. Meteor. Climatol., 52, 733–752, https://doi.org/10.1175/JAMC-D-12-0170.1

[perezcruz08] Perez-Cruz, F. (2008). Kullback-Leibler divergence estimation of continuous distributions. 2008 IEEE International Symposium on Information Theory, 1666-1670. https://doi.org/10.1109/ISIT.2008.4595271

[SR2004] Székely, G. J. and Rizzo, M. L. (2004) Testing for Equal Distributions in High Dimension, InterStat, November (5). https://www.researchgate.net/publication/228918499_Testing_for_equal_distributions_in_high_dimension

[RS2016] Rizzo, M. L., & Székely, G. J. (2016). Energy distance. Wiley Interdisciplinary Reviews: Computational Statistics, 8(1), 27–38. https://doi.org/10.1002/wics.1375

[AZ03] Aslan B. and Zech G. (2003) A new class of binning-free, multivariate goodness-of-fit tests: the energy tests. https://doi.org/10.48550/arXiv.hep-ex/0203010

PYTHON MODULE INDEX

INDEX