nctoolkit

Robert Wilson

Aug 31, 2023

QUICK OVERVIEW

nctoolkit is a comprehensive and computationally efficient Python package for analyzing and post-processing netCDF data.

CHAPTER

ONE

WHO IS NCTOOLKIT FOR?

Everyone from casual to regular users of netCDF data will find nctoolkit useful. Casual users will appreciate the easy ability to do such as things as matching up point observation data with gridded netCDF data. For expert users, nctoolkit provides the ability to carry out 80-100% of your day to day analysis and post-processing.

CHAPTER

TWO

WHAT TYPE OF DATA IS NCTOOLKIT DESIGNED FOR?

nctoolkit is designed primarily with climate and oceanic data in mind. If you work with this type of data, nctoolkit can help you do it quickly and efficiently.

CHAPTER

THREE

WHAT SYSTEMS CAN NCTOOLKIT WORK ON?

nctoolkit requires a Linux or macOS operating system.

CHAPTER

FOUR

WHAT CAN NCTOOLKIT DO?

The core abilities of nctoolkit include:

- Cropping to geographic regions
- Interactive plotting of data
- Subsetting to specific time periods
- Calculating time averages
- Calculating spatial averages
- Calculating rolling averages
- Calculating climatologies
- Creating new variables using arithmetic operations
- Calculating anomalies
- Horizontally and vertically remapping data
- Calculating the correlations between variables
- Calculating vertical averages for the likes of oceanic data
- Calculating ensemble averages
- Calculating phenological metrics

nctoolkit is developed as open source software by the Marine Systems Modelling group at Plymouth Marine Laboratory.

4.1 Installation

4.1.1 How to install nctoolkit

You will need a Linux or Mac operating system for nctoolkit to work. It will not work on Windows due to system requirements.

The best and easiest way to install nctoolkit is to use conda. This will install all system dependencies, and nctoolkit will just work out of the box. This can be done as follows:

```
$ conda install -c conda-forge nctoolkit
```

Conda can often install a very old version of nctoolkit. So you might want to install a recent version:

```
$ conda install -c conda-forge nctoolkit=0.9.4
```

Mamba is a smoother way to manage conda environments. If you don't use it, you should try. Install it from here.

Once mambaforge is installed you can install nctoolkit as follows:

```
$ mamba install -c conda-forge nctoolkit
```

This will be much faster to install than using conda, because mamba resolves environments much faster.

Note that recent releases are not available on macOS on conda. This issue is being investigated at the minute, and will hopefully be resolved shortly. In the meantime, if you are using macOS, it is best to install using pip.

If you do not use conda, you can install nctoolkit using pip. The package is available from the Python Packaging Index. To install nctoolkit using pip:

```
$ pip install nctoolkit
```

This will install the core dependencies of nctoolkit. If you want slightly better plots, you can install all dependencies:

```
$ pip install nctoolkit[complete]
```

Once you have installed nctoolkit using pip, you will need to install the system dependencies listed below.

To install the development version from GitHub:

```
$ pip install git+https://github.com/r4ecology/nctoolkit.git
```

4.1.2 How to get better looking plots

If you have installed the non-complete version of nctoolkit from pypi, you might also want to install cartopy to get better looking plots. This has some additional dependencies, so you may need to follow their guide here to ensure cartopy is installed fully.

If you installed nctoolkit using conda, you can install cartopy as follows:

```
$ mamba install -c conda-forge cartopy

$ conda install -c conda-forge cartopy
```

Cartopy is not installed by default because installation can be slow or difficult on pypi or conda. However, no such issues exist on mamba. So, **use mamba**.

4.1.3 Python dependencies

- Python (3.8 or later)
- numpy (1.14 or later)
- pandas (0.24 or later)
- xarray (0.14 or later)
- netCDF4 (1.53 or later)
- ncplot
- hvplot

- holoviews
- matplotlib
- bokeh
- panel
- metpy
- scipy

4.1.4 System dependencies

There are two main system dependencies: Climate Data Operators, and NCO. The easiest way to install them is using conda:

```
$ conda install -c conda-forge cdo

$ conda install -c conda-forge nco
```

or mamba:

```
$ mamba install -c conda-forge cdo

$ mamba install -c conda-forge nco
```

CDO is necessary for the package to work. NCO is an optional dependency and does not have to be installed.

If you are working on an Ubuntu system, you should be able to install CDO as follows:

```
$ sudo apt install cdo
```

If you want to install CDO from source, you can use one of the bash scripts available here.

4.2 Introduction to nctoolkit

nctoolkit is a multi-purpose tool for analyzing and post-processing netCDF files. It is designed explicitly with climate change and oceanographic work in mind. Under the hood, it uses Climate Data Operators (CDO), but it operates as a stand-alone package with no knowledge of CDO being required to use it.

Let's look at what it can do using a historical global dataset of sea surface temperature, which you can find here.

The preferred way to import nctoolkit is:

```
import nctoolkit as nc
```

4.2.1 It lets you quickly visualize data

nctoolkit offers plotting functionality that will let you automatically plot data from almost any type of netCDF file. It's as simple as the following, which calculates mean historical sea surface temperature and then plots it:

```
ds = nc.open_data("sst.mon.mean.nc")
ds.subset(year = 2000)
ds.plot()
```

4.2.2 It lets you easily subset data

If we want to look at a particular region, we can subset the data using the 'subset' method, and further select a particular year and month, we can do this as follows:

```
ds = nc.open_data("sst.mon.mean.nc")
ds.subset(year = 1998, month = 1, lon = [-13, 38], lat = [30, 67])
ds.plot()
```

4.2.3 It lets you calculate temporal averages

nctoolkit features a suite of methods, beginning with the letter t, that let you calculate temporal statistics. For example, if we wanted to calculate how much sea surface temperature varies each year, we could do this:

```
ds = nc.open_data("sst.mon.mean.nc")
ds.tmean()
ds.plot()
```

4.2.4 It lets you calculate spatial averages

Calculating the spatial average of a variable is as simple as:

```
ds = nc.open_data("sst.mon.mean.nc")
ds.spatial_mean()
ds.plot()
```

4.2.5 It lets you do mathematical operations

nctoolkit offers an 'assign' method for performing mathematical operations on variables. This works in a way that will be familiar to users of Pandas. The method is illustrated below in a processing chain that works out how much warmer each part of the ocean is than the global mean.

```
ds = nc.open_data("sst.mon.mean.nc")
ds.tmean()
ds.assign(delta = lambda x: x.sst - spatial_mean(x.sst), drop = True)
ds.plot("anomaly")
```

4.2.6 It lets you regrid data

nctoolkit has built-in methods for regridding data to user-specified grids. One of the most useful is *to_latlon*. This let's you regrid to a regular latlon grid. You just need to specify the extent of the new grid, the resolution and the regridding method.

```
ds = nc.open_data("sst.mon.mean.nc")
ds.subset(time = 0)
ds.to_latlon(lon = [-13, 38], lat = [30, 67], resolution = 1, method = "bilinear")
ds.plot()
```

4.2.7 It lets you calculate anomalies

In an example above we calculated the global mean sea surface temperature every month since 1850. But calculate the anomaly might be more interesting. The code below will calculate the change in global annual mean sea surface temperature since 1850-1969. The window argument let's you calculate it on a rolling basis.

```
ds = nc.open_data("sst.mon.mean.nc")
ds.spatial_mean()
ds.annual_anomaly(baseline = [1850, 1869], window= 20)
ds.plot()
```

4.2.8 It lets you calculate zonal averages

It is easy to calculate zonal averages using nctoolkit. In the example below change in temperature since 1850-1869 in each latitude band is calculated:

```
ds = nc.open_data("sst.mon.mean.nc")
ds.annual_anomaly(baseline = [1850, 1869], window= 20)
ds.zonal_mean()
ds.plot()
```

4.3 News

4.3.1 Release of v1.1.0

Versions 1.1.0 will be released in September 2023. This will be a major release with some improvements and new methods.

This release will require that CDO version 2.0.5 or above is installed.

The cost of maintaining support for CDO versions 1.9.10 and below was increasingly high compared with the declining user base for these versions.

A method for inverting vertical levels and latitude will be introduced: *invert*.

The *pub_plot* method will use slightly better colour scales for diverging scales.

4.3.2 Release of v1.0.0

Version 1.0.0 was released on 25th July 2023. This is a major release with some breaking changes.

A critical change is that accessing dataset attributes will force evaluation, i.e. *ds.run()* will occur if you access an attribute such as *ds.times*. This makes behaviour more consistent with what new users would expect.

This should not have a major impact on any coding workflows.

The *cdo_command* method has been changed so that it no longer checks the validity of the command before calling CDO. Checks can now be run using the *check* argument.

Some improvements have been made to method internals for *pub_plot*.

4.3.3 Release of v0.9.2

Version 0.9.1 will be released in April 2023.

This release will contain a new method for producing high quality static plots. Yuri Artioli of Plymouth Marine Laboratory contributed the core code to this new *pub_plot* method.

A new method, *set*, will be introduced that will make it easier to rename variables, and change units and long names etc.

Some improvements will be made to internals.

4.3.4 Release of v0.9.1

Version 0.9.1 was released on the 19th of April 2023. This was a quick release to deal with some breaking changes to add/subtract etc. methods due to the release of pandas 2.0.0.

4.3.5 Release of v0.9.0

Version 0.9.0 was released on 2nd March 2023. This is a major(ish) release with some breaking changes related to plotting.

On pypi, cartopy has been switched to an optional dependency because it was causing installation difficulties for some users. You can now do a "complete" installation using pip to get all optional dependencies:

```
$ pip install nctoolkit[complete]
```

This does not impact the conda version, which will behave as before.

Support is now available for Python 3.11.

File paths with spaces are now supported.

4.3.6 Release of v0.8.6

Version 0.8.6 was released on 23rd December 2022. This is a minor releases that tidies up some issues and has some method enhancements.

The *regrid* and *to_latlon* methods can now be more efficient for multi-file datasets where all files have the same grid. Previously, the methods identified the grids for all methods. You can now set the *one_grid* argument to *True*, which will result in the methods assuming all files have the same grid, and only the first file being checked.

There was an issue with multi-file datasets in parallel in Python 3.8 and 3.9. A confusing TypeError was being thrown due to signalling issues by multiprocessing. This gave the impression there was a problem with processing when there wasn't one. This problem is now fixed.

4.3.7 Release of v0.8.5

Version 0.8.5 was released on 14th December 2022. This is a minor release that deals with clean up issues on Jupyter notebooks. A change in a recent version of ipykernel was causing nctoolkit to not automatically remove temporary files on exit, though only in jupyter notebooks. This should now be fixed.

The *annual_anomaly* method now lets users temporally align the output, in the same way as other temporal methods such as *roll_mean*.

Some improvements have been made to internals for better warnings and errors.

4.3.8 Release of v0.8.4

Version 0.8.4 was released on 6th December 2022.

This update improves the ability to handle missing values. A method iss introduced for changing the fill value missing values, *set_fill*.

Another method *missing_as* is introduced. This will do the opposite of *as_missing*. Instead of setting a range of values to missing values, it will set missing values to a constant value.

Dataset contents will now show the fill value for variables. Furthermore, *open_data* will now check if the fill value is zero, which can cause problems for logical comparisons etc.

4.3.9 Release of v0.8.2

Version 0.8.2 was released on 25h November 2022. This release changed plotting so that it does not show coastlines by default.

Plotting with coastlines was causing plotting to crash on some systems due to issues with how nctoolkit's Python dependencies work with non-Python dependencies. Essentially plotting could crash if cartopy and pyproj were importable, but not fully functional. These were not a problem with nctoolkit installations from conda, which will install non-Python dependencies, but some non-conda insttallations would no longer plot maps as a Python dependency could be incompatible with the non-Python dependencies on user systems.

If you want to plot the coastline, do the following:

ds.plot(coast=True)

This is not an ideal fix, but it was necessary as a high proportion of users have a semi-functional cartopy installation, and there is no way for them to know that this is causing the plotting problem. A future release will hopefully provide automatic coastlines when cartopy and pyproj are fully functional on people's systems.

4.3.10 Release of v0.8.0

Version 0.8.0 was released on 17th November 2022. This was a major release that introduces some breaking changes.

The major improvement in this release is to vertical methods. All vertical methods should now work with files with vertical axes that are either consistent or vary spatially. Before some methods only worked with z-levels, i.e. files with fixed vertical levels. This change will result in a requirement that *vertical_mean*, *vertical_interp* and *vertical_integration* need users to specify whether the vertical levels are fixed spatially, using the fixed arg.

There were also some improvements to internals.

4.3.11 Release of v0.7.6

Release data: 30th September 2022.

This is a minor release that significantly simplifies basic arithmetic and logical operations.

Simple methods such as +, - etc. can now use standard python syntax.

For example, if you wanted to add 2 to a dataset you can now do the following:

ds.add(2)

as this instead

ds+2

The same goes for logical operators. You can do the following to identify if the values in a datset are below 2:

ds<2

whereas you previously had to do this:

ds.compare("<2")

Note: because nctoolkit methods only modify datasets and do not return datasets, the following will not work:

ds1+ds2+2

Instead, you would need to do:

ds1+ds2 ds1+2

4.3.12 Release of v0.7.1

Release data: 10th September 2022.

This is a major release with some breaking changes.

The deprecated *select* method has now been removed. Users should now use the *subset* method.

A progress bar will now display when processing large datasets. This will only show when nctoolkit thinks something will take a while. If you want to always show a progress bar for multi-file datasets, you can do this: *nc.options(progress = 'on')*.

4.3.13 Release of v0.6.0

Release date: 15th August 2022.

This is a major release that introduces some breaking changes. All methods that carry out temporal averaging of any sort will now align output times to the right. This applies to methods such as *tmean* and *rolling_mean*. The internals when *align = "left"* option have been modified, as the CDO call was sometimes giving incorrect results.

4.3.14 Release of v0.5.4

This is a minor release on August 10th 2022.

It improves the abilities of temporal methods, giving users the ability to select how they want times in output to be aligned.

For example, if you are calculating a rolling mean, you might want the output times to be the first, middle or final time in the temporal window. This release will add that ability to nctoolkit's temporal methods. Previously nctoolkit used CDO's default methods, and did not allow users to do anything else. By default, output dates will be aligned to the middle.

The *match_points* methods were throwing an error when there were non-unique vertical values. This is now fixed.

Some improvements have been made to package internals.

4.3.15 Release of v0.5.1

This was a minor release made on 30th June 2022. It includes method enhancements.

The *subset* method now allows negative time slicing.

The *set_missing* method is deprecated and replaced with a less ambiguously named *as_missing* method.

The *plot* method will no longer show a plot title by default to make things cleaner.

The *vertical_integration* method now works with multi-file datasets and will not calculate vertical integrations for the thickness variable.

Some improvements have been made to improve error messages, and the *check* method now checks for data type of time.

A new method `as_type` has been added for changing data type of individual variables and coordinates.

4.3.16 Release of v0.5.0

This relase was made on 13th June 2022. The *match_points* method now allows extrapolation to vertical depths.

4.3.17 Release of v0.4.9

This relase was made on 9th June 2022. The *subset* method now accepts levels.

4.3.18 Release of v0.4.8

This release improves temporal merging of large datasets. Previously on some systems this would fail on datasets made up of more than 1,000 files due to system limits. Under the hood, nctoolkit now deals with this.

The merge method also now contains a check argument that can be used to speed up merging of large datasets when you know the files can be merged problem-free. Previously, merge always checked if files being merged had the same variables when doing a temporal merge. This can now be switched off if you are confident this does not need to happen.

4.3.19 Release of v0.4.7

Version 0.4.7 was released on June 5th 2022.

This release contained a new method called match_points that can do matchups with a spatiotemporal dataframe.

4.3.20 Release of v0.4.6

Version 0.4.6 was released on June 3rd 2022.

This release will enhance existing methods.

The `select` method will be replaced by `subset`. This behave in the way same way as `select`, but will also allow users to subset data base on longitude and latitude using the `lon` and `lat` as args.

The export methods `to_nc`, `to_xarray` and `to_dataframe` now allow only a subset of the data to be exported. Additional arguments can be sent to the methods, which will then be sent to the `subset` method.

The new matchpoint methods for matching netCDF and point data have been smoothed out with additional options.

Minor bug fix: The weighted in datasets with recycled regridding weights were not copied properly. This is now fixed.

4.3.21 Release of v0.4.5

Version 0.4.5 was released in late May 2022. This was a minor release that fixed an issue with `ds.variables` when there were a) many variables and b) CDO version above 2.0.0.

4.3.22 Release of v0.4.4

Version 0.4.4 was released in late May 2022.

This version introduces a new class called *Matchpoint* which will allow automated matchups between netCDF files and point observations in pandas dataframes. This class is created using `nc.open_matchpoint`. Matchups are generated by using the `add_data`, `add_points`, `add_depths`, and `matchup` methods.

For datasets, `ds` now provides a more informative summary of dataset contents.

The `split` method now automatically sorts the files, so that they are sorted by date when temporal splitting occurs.

The methods `surface`, `merge_time` and `tvariance`` have been removed after periods of deprecation. Use `top`, `merge` and `tvar` instead.

4.3.23 Release of v0.4.3

Version 0.4.3 was released in May 2022. This is release with some new methods, improvements to internals some bug fixes. Code written for previous 0.4x versions of nctoolkit will be compatible.

This version will be compatible with CDO versions 2.0.5x.

A new function `open_geotiff` will allow GeoTiff files to be opened. This is a wrapper around rioxarray, which will convert the GeoTiff to NetCDF. It will require rioxarray to be installed.

A new method `surface_mask` has been added to enable identifying top levels with data in cases when there are missing values in the actual top level.

A new method `is_corrupt` has been added. This can identify whether NetCDF files are likely to be corrupt. Under-the hood, methods will now suggest running `is_corrupt` when system errors imply the files are corrupt.

The methods `to_xarray` and `to_dataframe` no long accept the *cdo_times* argument, as this has essentially been redundant for a few nctoolkit versions.

The `plot` method now lets users send kwargs to hvplot to make customizations, such as log-scales an option. This will require the latest version of ncplot.

The `select` method now lets user select days of month, using `ds.select(day = 1)`.

The `split` method now allows splitting by timestep using `split("timestep")`.

4.3.24 Release of v0.4.2

Version 0.4.2 was released in March 2022.

This is a minor release with a couple of method enhancements. Plots can now be saved to html files using the *out* arguments. The `nco_command` method now works over multiple cores when these are set using `nc.options`.

4.3.25 Release of v0.4.1

Version 0.4.1 was released in March 2022. This is a minor release focusing on improving nctoolkit internals.

A new method, called `check` is introduced that can be used to troubleshoot data problems and to ensure there are no obvious data issues (such as a lack of CF-compliance).

Users can now access dataset calendars using `ds.calendar`.

The `drop` method now lets you remove time steps using the `times` argument.

The dataset attribute *variables_detailed* is now removed after being replaced by *contents* in version 0.3.9.

This version will recommend CDO versions greater than 1.9.7, because ensuring nctoolkit compatibility with earlier versions was becoming difficult and likely of little need to users.

Some coding improvements have enhanced the performance of the `add`, `subtract` etc. methods.

Bug fixes: The methods `multiply` etc. failed when datasets did not have time as a dimension in version 0.4.0. This is now fixed. Previously, *ds.contents* always returned None for the number of time steps. Now fixed.

4.3.26 Release of v0.4.0

Version 0.4.0 was released in January 2022. This is a major release that features some breaking changes. Methods for adding, subtracting, multipling and substracting datasets from each other will be enhanced. Until now these methods used a simplistic approach values from matching time steps were added to each other, etc. So if you are subtracting a 12 time step file from a dataset, only the first 12 time steps were subtracted from. However, often this is not what you want. For example, you might want to subtract yearly months from a file which contains montly values for each year.

This version of nctoolkit updates these methods so that it can figure out what kind of addition etc. it should carry out. For example, if you have a dataset which has monthly values for each year from 1950 to 1999, and use `subtract` to subtract the values from a file which contains annual means for each year from 1950, it will subtract the annual mean for 1950 from each month in 1950 and the the annual mean for 1951 from each month in 1951, and so on.

Users are now able to specify the numeric precision of datasets using `ds.set_precision`. By default uses the underlying netCDF file's data type. This is normally not a problem. However, when the data type is integer, this can cause problems. `nc.open_data` has been updated with this issue in mind. It will now warn users when the data type of the netCDF is integer, and it suggested switching to float 'F64' or 'F32'.

The `drop` method has been enhanced. It now accepts day, month and year as arguments to enable dropping specific time periods. For example `ds.drop(month = 2, day = 29)` will remove leap days. Code written to use the old `drop` method will now fail, as keywords are now required.

The method `surface` has now been renamed `top` for consistency with `bottom`. `surface` is deprecated and will be removed in a few months.

The `split` method now allows users to split datasets into multiple files by variable.

`ds.times` now returns a datetime object, not a str as before.

4.3.27 Release of v0.3.9

Version 0.3.9 was released in November 2021. This is minor release focusing on under-the-hood improvements and new methods.

A new method, `from_xarray` is added for converting xarray datasets to nctoolkit datasets.

Methods for identifying how many missing values appear in datasets have been added: *na_count* and *na_frac*. These will identify the number or fraction of values that are missing values in each grid cell. The methods operate the same way as the temporal methods. So *ds.na_frac("year")* will result in what fraction of values are missing values each year.

Methods for better upscaling of datasets will be added: `box_mean`, `box_sum`, `box_max`. This will allow you to upscale to, for example, each 10 by 10 grid box using the mean of that grid box. This is useful for upscaling things like population data where you want the upscaled grid boxes to represent the entirety of the grid box, not the centre.

Improvements to `merge` have been made. When variables are not included in all files nctoolkit will now only merge those in each file in a multi-file dataset. Previously it threw an error.

Functions for finding the times and months in netCDF files are now available: `nc_years` and ``nc_months`.

The attribute `variables_detailed` has been changed to `contents`. It will also now give the number of time steps available for each variable.

`cdo_command` now allows users to specify whether the CDO command used is an ensemble method. Previously methods applied on a file by file basis.

4.3.28 Release of v0.3.8

Version 0.3.8 was released in October 2021. This is a minor release, focusing on under-the-hood improvements and introducing better handling of files with varying vertical layers.

A method, `vertical_integration` for calculating vertically integrated totals for netCDF data of the likes of oceanic data, where the vertical levels vary spatially, were introduced. `vertical_mean` has been improved and can now calculate vertical mean in cases where the cell thickness varies in space.

`merge_time` is deprecated, and its functionality will be incorporated into `merge`. So, following this release ensemble merging should use `merge`.

`open_url` is now able to handle multiple urls. Previously it could only handle one.

Some under-the-hood improvements have been made to `assign` to ensure that truth statements do not occassionally throw an error.

4.3.29 Release of v0.3.7

Version 0.3.7 was released in August 2021. This is a minor release.

New mathematical methods for simple operations on variables were added: `abs`, `power`, `square`, `sqrt`, `exp`, `log` and `log10`. These methods match numpy names.

Bug fixes: `assign` previously did not work with `log10`. Now fixed.

`compare_all` was deleted after a period of deprecation.

4.3.30 Release of v0.3.6

Version 0.3.6 was released in July 2021. This was a minor release.

New methods `ensemble_var` and `ensemble_stdev` were introduced for calculating variance and standard deviation across ensembles. The method `tvariance` will be deprecated and is now renamed `tvar` for naming consistency.

4.3.31 Release of v0.3.5

Version 0.3.5 was released in May 2021.

This is a minor release focusing on some under-the-hood improvements in performance and a couple of new methods.

It drops support for CDO version 1.9.3, as this is becoming too time-consuming to continue given the increasingly low reward.

A couple of new methods have been added. `distribute` enables files to be split up spatially into equally sized m by n rectangles. `collect` is the reverse of `distribute`. It will collect distributed data into one file.

In prior releases `assign` calls could not be split over multiple lines. This is now fixed.

There was a bug in previous releases where `regrid` did not work with multi-file datasets. This was due to the enabling of parallel processing with nctoolkit. The issue is now fixed.

The deprecated methods `mutate` and `assign` have now been removed. Variable creation should use `assign`.

4.3.32 Release of v0.3.4

Version 0.3.3 was released in April 2021.

This was a minor release focusing on performance improvements, removal of deprecated methods and introduction of one new method.

A new method `fill_na` has been introduced that allows missing values to be filled with the distanced weighted average.

The methods `remove_variables` and `cell_areas` have been removed and are replaced permanently by `drop` and `cell_area`.

4.3.33 Release of v0.3.2

Version 0.3.2 was released in March 2021. This was a quick release to fix a bug causing `to_nc` to not save output in the base directory.

4.3.34 Release of v0.3.1

Version 0.3.1 was released in March 2021. This is a minor release that includes new methods, under-the-hood improvements and the removal of deprecated methods.

New methods are introduced for identifying the first time step will specific numerical thresholds are first exceeded or fallen below etc: `first_above`, `first_below`, `last_above` and `last_below`. The thresholds are either single numbers or can come from a gridded dataset for grid-cell specific thresholds.

Methods to compare a dataset with another dataset or netCDF file have been added: `gt` and `lt`, which stand for 'greater than' and 'less than'.

Users are be able to recycle the weights calculated when interpolating data. This can enable much faster interpolation of multiple files with the same grid.

The temporal methods replaced by `tmean` etc. have now been removed from the package. So `monthly_mean` etc. can no longer be used.

4.3.35 Release of v0.3.0

Version 0.3.0 was released in February 2021. This will be a major release introducing major improvements to the package.

A new method `assign` is now available for generating new variables. This replaces the `mutate` and `transmute`, which were place-holder functions in the early releases of nctoolkit until a proper method for creating variables was put in place. `assign` operates in the same way as the `assign` method in Pandas. Users can generate new variables using lambda functions.

A major-change in this release is that evaluation is now lazy by default. The previous default of non-lazy evaluation was designed to make life slightly easier for new users of the package, but it is probably overly annoying for users to have to set evaluation to lazy each time they use the package.

This release features a subtle shift in how datasets work, so that they have consistent list-like properties. Previously, the files in a dataset given by the `` `current` `` attribute could be both a str or a list, depending on whether there was one or more files in the dataset. This now always gives a list. As a result datasets in nctoolkit have list-like properties, with `` `append` `` and `remove` methods available for adding and removing files. `remove` is a new method in this release. As before datasets are iterable.

This release will also allow users to run nctoolkit in parallel. Previous releases allowed files in multi-file datasets to be processed in parallel. However, it was not possible to create processing chains and process files in parallel. This is now possible in version thanks to under-the-hood changes in nctoolkit's code base.

Users are now able to add a configuration file, which means global settings do not need to be set in every session or in every script.

4.4 How to cite nctoolkit

If you have used this software in a scientific publication, we would appreciate citations to the following paper: Wilson, R.J., Artioli, Y., (2023). nctoolkit: A Python package for netCDF analysis and post-processing. Journal of Open Source Software, 8(88), 5494, https://doi.org/10.21105/joss.05494.

You can use the following BibTeX entry:

```
@article{Wilson2023, doi = {10.21105/joss.05494}, url = {https://doi.org/10.21105/joss.
↪05494}, year = {2023}, publisher = {The Open Journal}, volume = {8}, number = {88},␣
↪pages = {5494}, author = {Robert J. Wilson and Yuri Artioli}, title = {nctoolkit: A␣
↪Python package for netCDF analysis and post-processing}, journal = {Journal of Open␣
↪Source Software} }
```

4.5 Datasets

4.5.1 Data format requirements

nctoolkit requires NetCDF data that follow the GDT, COARDS or CF Conventions. Its computational backend is CDO, which be able to carry out most operations regardless of whether it is compliant with those conventions. In general, most data producers follow CF-conventions when generating NetCDF files, however if you are unclear if you are working with compliant files you can check here.

4.5.2 Opening datasets

There are 3 ways to create a dataset: `open_data`, `open_url` or `open_thredds`.

If the data you want to analyze is available on your computer use `open_data`. This will accept either a path to a single file or a list of files. It will also accept wildcards.

If you want to use data that can be downloaded from a url, just use `open_url`. This will download the netCDF files to a temporary folder, and it can then be analyzed.

If you want to analyze data that is available from a thredds server or OPeNDAP, then use `open_thredds`. The file paths should end with .nc.

[1]:
```
import nctoolkit as nc
```

```
nctoolkit is using the latest version of Climate Data Operators version: 2.0.5
```

If you want to get a quick overview of the contents of a dataset, we can use the `contents` attribute. This will display a dataframe showing the variables available in the dataset and details about the variable, such as the units and long names. The example below opens a sea-surface temperature dataset and displays the contents.

[2]:
```
ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.ltm.1981-
↪2010.nc")
ds
```

[2]:
```
<nctoolkit.DataSet>:
Number of files: 1
File contents:
         variable  ntimes  npoints  nlevels                                                ␣
↪          long_name  unit data_type
0              sst      12    64800        1  Long Term Mean Monthly Means of Global Sea␣
↪Surface Temperature  degC       F32
1  valid_yr_count      12    64800        1                          count of non-missing␣
↪values used in mean  None       I16
```

4.5.3 Checking validity of source data

nctoolkit should work out of the box with most NetCDF data. However, it is possibly the format of the data could be incompatible with the system libraries used by nctoolkit or the files could be corrupt. To carry out a general check on the data use the `check` method as follows:

[ ]:
```
ds.check()
```

```
****************************************
Checking data types
****************************************
The variable I16 has integer data type. Consider setting data type to float 'F64' or 'F32
↪' using set_precision.
****************************************
Checking time data type
****************************************
****************************************
Running CF-compliance checks
****************************************
Issue with variable: sst
------------------
ERROR: Invalid attribute name: _ChunkSizes

------------------
****************************************
Checking grid consistency
****************************************
```

This will carry out some basic checks on data format compatability. You should install the cfchecker package if you want `check` to check for CF-compliance.

If you want to check if the files in a dataset are corrupt, the following should tell you. This will simply read and write the data in the source files to a temporary file, which should be sufficient to ensure files are not corrupt.

[ ]:
```
ds.is_corrupt()
```

4.5.4 Modifying datasets

If you want to modify a dataset, you just need to use nctoolkit's built in methods. These methods operate directly on the dataset itself. The example below selects the first time step in a sea surface temperature dataset and plots the result.

```
[ ]: ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.ltm.1981-
     ↪2010.nc")
     ds.subset(time = 0)
     ds.plot()
```

Underlying datasets are temporary files representing the current state of the dataset. We can access this using the `current` attribute:

```
[ ]: ds.current
```

In this case, we have a single temporary file. Any temporary files will be generated and deleted, as needed, so there should be no need to manage them yourself.

4.5.5 Lazy evaluation by default

Look at the processing chain below.

```
[ ]: ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.ltm.1981-
     ↪2010.nc")
     ds.assign(sst = lambda x: x.sst + 273.15)
     ds.subset(months = 1)
     ds.subset(lon = [-80, 20], lat = [30, 70])
     ds.spatial_mean()
```

What is potentially wrong with this? It carries out four operations, so we absolutely do not want to create temporary file in each step. So instead of evaluating the operations line by line, nctoolkit only evaluates them either when you tell it to or it has to. So in the code example above we have told, nctoolkit what to do to that dataset, but have not told it to actually do any of it.

We can see this if we look at the current state of the dataset. It is still the starting point:

```
[ ]: ds.current
```

If we want to evaluate this we can use the `run` method or methods such as `plot` that require commands to be evaluated.

```
[ ]: ds.run()
     ds.current
```

This method chaining ability within nctoolkit comes from Climate Data Operators (CDO), which is the backend computational engine for nctoolkit. nctoolkit does not require you to understand CDO, but if you want to see the underlying CDO commands used, just use the `history` attribute. In the example, below, you can see that 4 lines of Python code have been converted to a single CDO command.

```
[ ]: ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.ltm.1981-
     ↪2010.nc")
     ds.assign(sst = lambda x: x.sst + 273.15)
     ds.subset(months = 1)
     ds.subset(lon = [-80, 20], lat = [30, 70])
     ds.spatial_mean()
     ds.history
```

Then if we run this, we can see the full command used:

```
[ ]: ds.run()
ds.history
```

If you want to visualize a dataset, you just need to use `plot`:

```
[ ]: ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.ltm.1981-
↪2010.nc")
ds.subset(time = 0)
ds.plot()
```

4.5.6 Method chaining

When you start to use nctoolkit it is important to realize that it does not allow method chaining in the way pandas and xarray do. So the following will not work:

```
[ ]: (
    ds
    .tmean()
    .spatial_mean()
    .add(1)
)
```

This is because this type of method chaining requires the methods to return an object. However, nctoolkit's methods in general do not return objects. Instead they modify them.

You would need to do the following instead:

```
[ ]: ds.tmean()
ds.spatial_mean()
ds.add(1)
```

4.5.7 Dataset attributes

You can find out key information about a dataset using its attributes. If you want to know the variables available in a dataset called ds, we would do:

```
[ ]: ds.variables
```

If you want more details about the variables, access the `contents` attribute. This will tell you details such as long names, units, number of time steps etc. for each variable.

```
[ ]: ds.contents
```

If you want to know the vertical levels available in the dataset, we use the following.

```
[ ]: ds.levels
```

If you want to know the files in a dataset, we would do this. nctoolkit works by generating temporary files, so if you have carried out any operations, this will show a list of temporary files.

```
[ ]: ds.current
```

If you want to find out what times are in the dataset we do this:

```
[ ]: ds.times
```

If you want to find out what months are in the dataset:

```
[ ]: ds.months
```

If you want to find out what years are in the dataset:

We can also access the history of operations carried out on the dataset. This will show the operations carried out by nctoolkit's computational back-end CDO:

```
[ ]: ds.history
```

4.6 Data supported

nctoolkit will support analysis of most netCDF data. However, there are some limitations and requirements. Most operations in nctoolkit rely on Climate Data Operators (CDO) to perform the heavy lifting. CDO requires that files have at most 4 dimensions, which should be longitude and latitude, and time and depth/height. It provides support for structured grids such as regular lon/lat or curvilinear grids, and unstructured grids.

4.6.1 Horizontal grids

nctoolkit will work with more or less any structured horizontal grid, so long as it follows the GDT, COARDS or CF conventions.

It provides limited supported for unstructured horizontal grids. These grids are often idiosyncratic and require special treatment and therefore only limited functionality is available in CDO. However, interpolation via the nearest neighbour method is supported. In some cases, the metadata of the netCDF file may need to be changed to allow CDO to work with the data. So, if you are working with unstructured data and running into problems, reach out to us at the nctoolkit Discussions page).

In some cases methods may not be able to provide a fully accurate answer due to deficiencies in the underlying file metadata. For example, the *spatial_mean* method needs to be able to calculate the area of each grid cell, but sometimes data providers fail to include this information in their files. In such cases, warning messages will be printed to the screen, but you can reach out to us if you want a fix for data problems.

4.6.2 Vertical grids

nctoolkit provides support for vertical grids that have either consistent or varying horizontal levels. So, in almost all cases it will work with your data. Occasionally, you may run into problems due to deficiencies in the raw netCDF files. For example, for vertical averaging to be totally accurate, the thickness of each vertical level is required. However, sometimes files do not contain this information, and there is no way to infer it. At present, nctoolkit is focused on analyzing data, not correcting, and therefore is not designed to fix issues in the raw files. However, if you run into any contact us here) and we can help you.

4.6.3 The time axis

So long as your time axis is CF-compliant, nctoolkit should have no problem handling it. However, CDO requires only one time axis. If you have multiple time axes, it will pick one. This is almost never an issue, unless you have very idiosyncratic time axes.

4.6.4 Data types

nctoolkit supports all data types that CDO supports. This includes 32-bit and 64-bit floating point numbers, and 8-bit, 16-bit and 32-bit integers. By default CDO, and therefore nctoolkit, will use the data type of the netCDF file for any computations. In general, this is not an issue. However, some times you need to be careful when you are working with files with integer data formats. These may need to be changed to float using the *set_precision* method.

Similarly, you may run into rare problems due to poorly defined netCDF files that cause computational problems. For example, netCDF files can have poorly defined maximum values that result in errors when carrying out simple calculations.

4.6.5 How to check if your files are CF compliant

If you are unsure if your files are CF compliant, you can check them using the CF checker. This is a great tool that will check your files for compliance with the CF conventions. If this throws any errors, then you might have issues analyzing the file with nctoolkit. An error would imply that it is not possible for CDO to figure out the structure of the file, and therefore it will not be able to perform certain operations on it. Most of the time you can fix these problems with CDO or NCO, so reach out to us if you need help.

4.6.6 How to deal with data problems

nctoolkit is designed primarily as a data analysis package. At present, it provides minimal functionality for fixing problems in the raw data, especially in the metadata. This is expected to change in future releases. However, until then it is best to reach out to us if you run into problems. These can typically be solved using either NCO or CDO.

4.7 Importing and exporting data

nctoolkit can work with data available on local file systems, urls and over thredds and OPeNDAP.

4.7.1 Opening single files and ensembles

If you want to import a single netCDF file as a dataset, do the following:

```
import nctoolkit as nc
ds = nc.open_data(infile)
```

The *open_data* function can also import multiple files. This can be done in two ways. If we have a list of files we can do the following:

```
import nctoolkit as nc
ds = nc.open_data(file_list)
```

Alternatively, *open_data* is capable of handling wildcards. So if we have a folder called data, we can import all files in it as follows:

```
import nctoolkit as nc
ds = nc.open_data("data/*.nc")
```

4.7.2 Opening files from urls/ftp

If we want to work with a file that is available at a url or ftp, we can use the *open_url* function. This will start by downloading the file to a temporary folder, so that it can be analysed.

```
import nctoolkit as nc
ds = nc.open_url(www.foo.nc)
```

4.7.3 Opening data available over thredds servers or OPeNDAP

If you want to work with data that is available over a thredds server or OPeNDAP, you can use the *open_thredds* method. This will require that the url ends with ".nc".

```
import nctoolkit as nc
ds = nc.open_thredds(www.foo.nc)
```

4.7.4 Exporting datasets

nctoolkit has a number of built in methods for exporting data to netCDF, pandas dataframes and xarray datasets.

4.7.5 Save as a netCDF

The method `to_nc` lets users export a dataset to a netCDF file. If you want this to be a zipped netCDF file use the `zip` method before to `to_nc`. An example of usage is as follows:

```
ds = nc.open_data(infile)
ds.tmean()
ds.zip()
ds.to_nc(outfile)
```

4.7.6 Convert to pandas dataframe

The method `to_dataframe` lets users export a dataset to a pandas dataframe.

```
ds = nc.open_data(infile)
ds.tmean()
df = ds.to_dataframe()
```

4.7.7 Interacting with xarray datasets

If you want to move between nctoolkit and xarray dataset, you can use `from_xarray` and `to_xarray`.

The method `to_xarray` lets users export a dataset to an xarray dataset. An example of usage is as follows:

```
ds = nc.open_data(infile)
ds.tmean()
xr_ds = ds.to_xarray()
```

If you want to convert an xarray dataset to an nctoolkit dataset, you can just the `from_xarray` function, as follows:

```
ds = nc.from_xarray(ds_xr)
```

4.7.8 Exporting subsets of data

If you want to only export a subset of the data you can do this by providing additional args to the `to_nc`, `to_xarray` and `to_dataframe` methods. These args will then be sent to the `subset` method.

For example, if you only wanted to export the year 2000 to xarray, you would do the following:

```
ds.to_xarray(year = 2000)
```

Or if you wanted a spatial subset of the data you could do this:

```
ds.to_xarray(lon = [0, 90], lat = [0, 90])
```

4.8 Subsetting data

nctoolkit has many built in methods for subsetting data. The main method is `subset`. This let's you select specific variables, years, months, seasons and timesteps.

4.8.1 Selecting variables

If you want to select specific variables, you would do the following:

```
ds.subset(variables = ["var1", "var2"])
```

If you only want to select one variable, you can do this:

```
ds.subset(variables = "var1")
```

4.8.2 Selecting years

If you want to select specific years, you can do the following:

```
ds.subset(years = [2000, 2001])
```

Again, if you want a single year the following will work:

```
ds.subset(years = 2000)
```

The `select` method allows partial matches for its arguments. So if we want to select the year 2000, the following will work:

```
ds.subset(year = 2000)
```

In this case we can also select a range. So the following will work:

```
ds.subset(years = range(2000, 2010))
```

4.8.3 Selecting months

You can select months in the same way as years. The following examples will all do the same thing:

```
ds.subset(months = [1,2,3,4])
ds.subset(months = range(1,5))
ds.subset(mon = [1,2,3,4])
```

4.8.4 Selecting seasons

You can easily select seasons. For example if you wanted to select winter, you would do the following:

```
ds.subset(season = "DJF")
```

4.8.5 Selecting timesteps

You can select specific timesteps from a dataset in a similar manner. For example if you wanted to select the first two timesteps in a dataset the following two methods will work:

```
ds.subset(time = [0,1])
ds.subset(time = range(0,2))
```

4.8.6 Geographic subsetting

If you want to select a geographic subregion of a dataset, you can use *subset*. This method will select all data within a specific longitude/latitude box. You just need to supply the minimum longitude and latitude required. In the example below, a dataset is cropped with longitudes between -80 and 90 and latitudes between 50 and 80:

```
ds.subset(lon = [-80, 90], lat = [50, 80])
```

4.9 Interpolation

nctoolkit features built in methods for horizontal and vertical interpolation.

4.9.1 Horizontal interpolation

We will illustrate how to carry out horizontal interpolation using a global dataset of global SST from NOAA. Find out more information about the datset here.

The data is available using a thredds server. So we will work with the first time step, which looks like this:

```
import nctoolkit as nc
ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.mean.nc")
ds.subset(time = 0)
ds.plot()
```

4.9.2 Interpolating to a regular lonlat grid

If you want to interpolate to a regular latlon grid, you can use `to_latlon`. `lon` and `lat` specify the minimum and maximum longitudes and latitudes, while `res`, a 3 variable list specifies the resolution. For example, if we wanted to regrid the globe to 0.5 degree north-south by 1 degree east-west resolution, we could do the following:

```
ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.mean.nc")
ds.subset(timestep = 0)
ds.to_latlon(lon = [-79.5, 79.5], lat = [0.75, 89.75], res = [1, 0.5])
ds.plot()
```

4.9.3 Interpolating to another dataset's grid

If we are working with two datasets and want to put them on a common grid, we can interpolate one onto the other's grid. We can illustate this with a dataset of global sea surface temperature. Let's start by cropping this dataset to the northern hemisphere.

```
ds1 = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.mean.nc
↪")
ds1.subset(timestep = 0)
ds1.subset(lat = [0, 90])
ds1.plot()
```

Now, we can regrid the original file to this northern hemisphere grid.

```
ds2 = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.mean.nc
↪")
ds2.subset(timestep = 0)
ds2.regrid(ds1)
ds2.plot()
```

This method will also work using netCDF files. So, if you wanted you can also use a path to a netCDF file as the target grid.

4.9.4 How to reuse the weights for regridding

Under the hood nctoolkit regrids data by first generating a weights file. There are situations where you will want to be able to re-use these weights. For example, if you are post-processing a large number of files one after the other. To make this easier nctoolkit let's you recycle the regridding info. This let's you interpolate using either `regrid` or `to_latlon`, but keep the regridding data for future use by `regrid`.

```
ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.mean.nc")
ds.subset(timestep = 0)
ds.to_latlon(lon = [-79.5, 79.5], lat = [-0.75, 89.75], res = [1, 0.5], recycle = True)
ds.plot()
```

```
ds1 = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.mean.nc
↪")
ds1.subset(timestep = 0)
ds1.regrid(ds)
ds1.plot()
```

4.9.5 Interpolating to a set of coordinates

If you want to regrid a dataset to a specified set of coordinates you can `regrid` and a pandas dataframe. The first column of the dataframe should be the longitudes and the second should be latitudes. The example below regrids a sea-surface temperature dataset to a single location with longitude -30 and latitude 50.

```
import pandas as pd
ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.mean.nc")
ds.subset(timestep = 0)
coords = pd.DataFrame({"lon":[-30], "lat":[50]})
ds.regrid(coords)
ds.to_dataframe()
```

This results in a dataframe with the following values:

4.9.6 Horizontal Resampling

If you want to make data more coarse spatially, just use the `resample_grid` method. This will, for example, let you select every 2nd grid grid cell in a north-south and east-west direction. This is illustrated in the example below, where a dataset which has spatial resolution of 1 by 1 degrees is coarsened, so that only every 10th cell is selected in a north-south and east-west. In other words it is now a 10 degrees by 10 degrees dataset.

```
ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.mean.nc")
ds.subset(timestep = 0)
ds.resample_grid(10)
ds.plot()
```

4.9.7 Spatial infilling

Some times you will have data with missing values, which you want to replace with a nearby value. nctoolkit handles this situation using the `fill_na` method. This uses distance-weighting. You just need to specify the number of nearest-neighbours to use for the weighting. For example, if you simply want to replace missing values with their nearest neighbour, you just set the number to 1, as follows:

```
ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.mean.nc")
ds.subset(timestep = 0)
ds.fill_na(1)
ds.plot()
```

4.9.8 Vertical interpolation

We can carry out vertical interpolation using the `vertical_interp` method. This is particularly useful for oceanic data. This is illustrated below by interpolating depth-resolved ocean temperatures from NOAA's World Ocean Atlas for January to a depth of 500 metres. The `vertical_interp` method requires a `levels` argument, which is sea-depth in this case.

```
ds = nc.open_thredds("https://www.ncei.noaa.gov/thredds/dodsC/ncei/woa/temperature/decav/
↪1.00/woa18_decav_t00_01.nc")
ds.subset(timestep = 0)
ds.vertical_interp(levels = 500, fixed = True)
ds.plot()
```

4.10 Plotting

nctoolkit provides automatic plotting of netCDF data in a similar way to the command line tool ncview.

If you have a dataset, simply use the `plot` method to create an interactive plot that matches the data type.

We can illustate this using a sea surface temperature dataset available here.

Let's start by calculating mean sea surface temperature for the year 2000 and plotting it:

```
import nctoolkit as nc
ff =  "sst.mon.mean.nc"
ds = nc.open_data(ff)
```

(continues on next page)

(continued from previous page)

```
ds.subset(year = 2000)
ds.plot()
```

We might be interested in the zonal mean. nctoolkit can automatically plot this easily:

```
ff =  "sst.mon.mean.nc"
ds = nc.open_data(ff)
ds.subset(year = 2000)
ds.tmean()
ds.zonal_mean()
ds.plot()
```

nctoolkit can also easily handle heat maps. So, we can easily plot the change in zonal mean over time:

```
ff =  "sst.mon.mean.nc"
ds = nc.open_data(ff)
ds.zonal_mean()
ds.annual_anomaly(baseline = [1850, 1869], window = 20)gg
ds.plot()
```

In a similar vein, it can automatically handle time series. Below we plot a time series of global mean sea surface temperature since 1850:

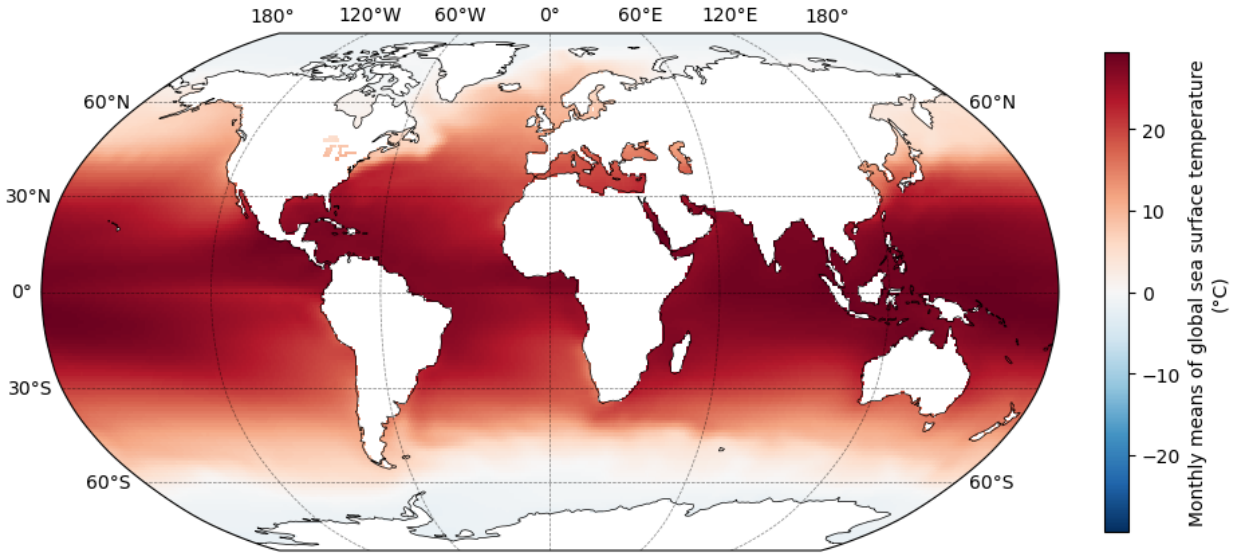
```
ff =  "sst.mon.mean.nc"
ds = nc.open_data(ff)
ds.spatial_mean()
ds.plot()
```

4.10.1 Publication quality plots

The ability to produce plots that are at or close to publication quality was introduced in version 0.9.2 in nctoolkit. This is carried out using the `pub_plot` method, and is designed to quickly produce a plot that is suitable for a publication or presentation.

```
ff =  "sst.mon.mean.nc"
ds = nc.open_data(ff)
ds.tmean()
ds.pub_plot()
```

display a png file called pub_plot1.png



Currently, `pub_plot` is restricted to work with regular lon/lat grids. A limited number of modifications can be made, such as changes to the colour scale. Over-time this will become a fully featured method.

4.10.2 Plotting internals

Plotting is carried out using the ncplot package. ncplot will look at the dataset and identify a suitable plotting method. This is carried out internally using hvplot. If you come across any errors, please raise an issue here.

This is a package that aims to deliver plotting for rapid exploratory analysis, and therefore it does not offer a large number of customizations. However, because it is built on hvplot, you can use most of the customization options available in hvplot, which are detailed here. Arguments such as *title*, *logz* and *clim* can be passed to *plot* and will be automatically passed to the hvplot method used .

4.11 Temporal statistics

nctoolkit has a number of built-in methods for calculating temporal statistics, all of which are prefixed with t: `tmean`, `tmin`, `tmax`, `trange`, `tpercentile`, `tmedian`, `tvariance`, `tstdev` and `tcumsum`.

These methods allow you to quickly calculate temporal statistics over specified time periods using the `over` argument.

By default the methods calculate the value over all time steps available. For example the following will calculate the temporal mean:

```
import nctoolkit as nc
ds = nc.open_data("sst.mon.mean.nc")
ds.tmean()
```

However, you may want to calculate, for example, an annual average. To do this we use `over`. This is a list which tells the function which time periods to average over. For example, the following will calculate an annual average:

```
ds.tmean(["year"])
```

If you are only averaging over one time period, as above, you can simply use a character string:

```
ds.tmean("year")
```

The possible options for `over` are "day", "month", "year", and "season". In this case "day" stands for day of year, not day of month.

In the example below we are calculating the maximum value in each month of each year in the dataset.

```
ds.tmax(["month", "year"])
```

4.11.1 Calculating rolling averages

nctoolkit has a range of methods to calcate rolling averages: `rolling_mean`, `rolling_min`, `rolling_max`, `rolling_range` and `rolling_sum`. These methods let you calculate rolling statistics over a specified time window. For example, if you had daily data and you wanted to calculate a rolling weekly mean value, you could do the following:

```
ds.rolling_mean(7)
```

If you wanted to calculated a rolling weekly sum, this would do:

```
ds.rolling_sum(7)
```

4.11.2 Calculating anomalies

nctoolkit has two methods for calculating anomalies: `annual_anomaly` and `monthly_anomaly`. Both methods require you to specify a baseline period to calculate the anomaly against. They require that you specify a baseline period showing the minimum and maximum years of the climatological period to compare against.

So, if you wanted to calculate the annual anomaly compared with a baseline period of 1950-1969, you would do this:

```
ds.annual_anomaly(baseline = [1950, 1969])
```

By default, the annual anomaly is calculated as the absolute difference between the annual mean in a year and the mean across the baseline period. However, in some cases this is not suitable. Instead you might want the relative change. In that case, you would do the following:

```
ds.annual_anomaly(baseline = [1950, 1969], metric = "relative")
```

You can also smooth out the anomalies, so that they are calculated on a rolling basis. The following will calculate the anomaly using a rolling window of 10 years.

```
ds.annual_anomaly(baseline = [1950, 1969], window = 10)
```

Monthly anomalies are calculated in the same way:

```
ds.monthly_anomaly(baseline = [1950, 1969]
```

Here the anomaly is the difference between the value in each month compared with the mean in that month during the baseline period.

4.11.3 Calculating climatologies

This means we can easily calculate climatologies. For example the following will calculate a seasonal climatology:

```
ds.tmean("season")
```

These methods allow partial matches for the arguments, which means you do not need to remember the precise argument each time. For example, the following will also calculate a seasonal climatology:

```
ds.tmean("Seas")
```

Calculating a climatological monthly mean would require the following:

```
ds.tmean("month")
```

and daily would be the following:

```
ds.tmean("day")
```

4.11.4 Calculating climatologies

This means we can easily calculate climatologies. For example the following will calculate a seasonal climatology:

```
ds.tmean("season")
```

4.11.5 Cumulative sums

We can calculate the cumulative sum as follows:

```
ds.tcumsum()
```

Please note that this can only calculate over all time periods, and does not accept an `over` argument.

4.12 Multi-file datasets

nctoolkit is built to handle multi-file datasets easily and efficiently. Parallel processing of files, ensemble averaging and merging are all easily done.

To create a multi-file dataset, you just need to supply a list of files to `open_data`. Alternatively, you can use wild cards. The following will create a multi-file dataset with all of the files in the foo folder:

```
import nctoolkit as nc
ds = nc.open_data("foo/*.nc")
```

Standard nctoolkit methods can then be applied to each file within the ensemble. For example, if we wanted a temporal mean of each file, we would do the following:

```
ds.tmean()
```

Note, to avoid any confusion: this operation will only apply to individual members of the multi-file dataset. We will later discuss ensemble methods such as `ensemble_mean`, which let you calculate statistics across the ensemble.

4.12.1 Merging multi-file datasets

There are two ways to merge mult-file datasets, time-based and variable-based.

Merging by time is done as follows:

```
ds.merge("time")
```

This will join files together so that their times join up. It should be used when files have the same variables and grids, but distinct times.

The second merging method is joining variables. In this case files should have the same time steps or one file should have at most one time step. This is done as follows:

```
ds.merge("variable")
```

By default, nctoolkit uses variable-based merging.

4.12.2 Speeding up multi-file processing

If you have access to multiple cores, it is very easy to ensure files within a multi-file dataset are processed in parallel. Just set the number of cores to be used. In the following case, we set it to 6:

```
nc.options(cores = 6)
```

This results in files being processed simultaneously with 6 cores.

If you are working on multi-file datasets, it is almost always much faster to set the number of cores to a high number and carry out operations on the files before merging them using `merge` and not the other way round.

4.12.3 Ensemble statistics

In some cases, you will want to calculate averages etc. across the multi-file dataset. For example, each file in a dataset could be from a different climate model and you might simply the mean value across them. This is very easily done. We can just calculate the ensemble mean as follows:

```
ds.ensemble_mean()
```

This will calculate the mean for each time step. For example, if you have an ensemble which is made of monthly projections of temperature from 20 different climate models, `ensemble_mean` will calculate the monthly mean of those 20 models.

Multiple ensemble methods are available: `ensemble_mean`, `ensemble_percentile`, `ensemble_stdev`, `ensemble_var`, `ensemble_max`, `ensemble_min`, `ensemble_range` and `ensemble_sum`.

4.13 Matchups with point data

A common challenge when working with netCDF data is matching up with point data. This is often difficult because point data is sparse both spatially and temporally, and when working in the ocean this data can be at varying depths. From version 0.4.7 on, nctoolkit includes the ability to match datasets to spatiotemporal dataframes. Here we will provide an overview of how to do this.

4.13.1 Matching data at specific locations

First, we will illustrate how matchpoint works for data at specific spatial locations and depths. After this we will deal with different times. The data will be ocean nitrate from NOAA's World Ocean Atlas.

We can download part of it as follows:

[1]:
```
import nctoolkit as nc
ds = nc.open_thredds('https://data.nodc.noaa.gov/thredds/dodsC/ncei/woa/nitrate/all/1.00/
↪woa18_all_n01_01.nc', checks = False)
ds.crop(lon =  [-40, 20], lat = [40, 70], nco = True)
ds.subset(variables = "n_an")
ds.run()
```

```
nctoolkit is using Climate Data Operators version 1.9.10
```

This is a subset of the data covering a large part of the North Atlantic, and it has nitrate values from the sea surface to the sea floor.

[2]:
```
ds.plot()
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

```
/home/robert/miniconda3/envs/notebook/lib/python3.9/site-packages/ncplot/plot.py:181:␣
↪UserWarning: Warning: xarray could not decode times!
```

[2]:
```
:DynamicMap   [depth]
   :Overlay
      .Image.I     :Image   [lon,lat]   (n_an)
      .Coastline.I :Feature   [Longitude,Latitude]
```

Now, let's say we had the following dataframe of 4 coordinates and depths. How would we identify the nitrate values using nctoolkit?

[3]:
```
import pandas as pd
df = pd.DataFrame({"lon":[-10, -12, -14, -16], "lat":[45, 50, 53, 55], "depth":[4, 2, 30,
↪ 40]})
df
```

[3]:

	lon	lat	depth
0	-10	45	4
1	-12	50	2
2	-14	53	30
3	-16	55	40

Note: if we are matching datasets to dataframes, the dataframe columns must be named one of the following: 'lon', 'lat', 'depth', 'year', 'month' or 'day'.

If we want to match our dataset to this dataframe we use the `match_points` method as follows:

[4]:
```
ds.match_points(df)
```

```
Depths assumed to be [0.0, 5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0, 45.0, 50.0, 55.
↪0, 60.0, 65.0, 70.0, 75.0, 80.0, 85.0, 90.0, 95.0, 100.0, 125.0, 150.0, 175.0, 200.0,␣
↪225.0, 250.0, 275.0, 300.0, 325.0, 350.0, 375.0, 400.0, 425.0, 450.0, 475.0, 500.0,␣
↪550.0, 600.0, 650.0, 700.0, 750.0, 800.0]
All variables will be used
Points will be matched for all time steps
```

[4]:

	lon	lat	depth	n_an	day	month	year
0	-10	45	4	5.661312	16	1	1958
1	-12	50	2	8.932839	16	1	1958
2	-14	53	30	8.672163	16	1	1958
3	-16	55	40	6.973096	16	1	1958

We now have the matchups required. The `match_points` method returns a pandas `dataframe` with the desired matchups.

You will get messages from nctoolkit confirming some of the assumptions taken when matching up. In most cases these can be ignored. The only exception is with depths. nctoolkit will derive from these from the dataset, but some times this will not be appropriate. Just keep an eye out for the message and explicitly provide depths if necessary.

4.13.2 Spatial matchup approach

The approach taken to matching up data spatially is as follows. First, data is regridded horizontally using bilinear interpolation to the lon/lat pairs provided. If depths are provided the data is than interpolated vertically with 1d interpolation using scipy.

4.13.3 Spatiotemporal matchups

We will now illustrate how to do spatiotemporal matchups. This will be done with air temperature from the CMIP6 climate model GFDL-CM4. This is a large file, but it can be downloaded by clicking here. The dataset contains gridded daily air temperature for the earth between 1850 and 1859.

Let's start by matching it up with the following dataframe:

[5]:
```
import nctoolkit as nc
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
df = pd.DataFrame({"lon": [50, 60], "lat": [50, 45], "year":[1850, 1852], "month":[1, 3],
↪ "day":[2, 3]})
df
```

[5]:

	lon	lat	year	month	day
0	50	50	1850	1	2
1	60	45	1852	3	3

This only contains two data points, but for different times. We can match up our dataset as before:

[6]:
```
ds = nc.open_data("tas_day_GFDL-CM4_historical_r1i1p1f1_gr1_18500101-18691231.nc",␣
↪checks = False)
df_match = ds.match_points(df)
df_match
```

```
All variables will be used
```

[6]:

	lon	lat	tas	year	month	day
0	50.0	50.0	252.836945	1850	1	2
1	60.0	45.0	271.053040	1852	3	3

As expected, we now have a pandas dataframe with the surface air temperature for the locations and times specified.

The `match_points` method works in a similar way to the pandas `merge` method. So, if we only specified year and month, and ignore day, we would get every day for those years and months, as follows:

[7]:
```
df_match = ds.match_points(df.drop(columns = "day"))
df_match
```

```
All variables will be used
```

[7]:

	lon	lat	tas	year	month	day
0	50.0	50.0	254.234680	1850	1	1
1	50.0	50.0	252.836945	1850	1	2
2	50.0	50.0	252.467865	1850	1	3
3	50.0	50.0	253.731049	1850	1	4
4	50.0	50.0	245.843506	1850	1	5
..	...	...	...	...	...	...
57	60.0	45.0	276.231720	1852	3	27
58	60.0	45.0	277.647888	1852	3	28
59	60.0	45.0	275.756226	1852	3	29
60	60.0	45.0	274.968018	1852	3	30
61	60.0	45.0	277.621979	1852	3	31

[62 rows x 6 columns]

We now have each day for the given times.

4.13.4 Optional arguments

The `match_points` method provided optional arguments that can refine the matchup process. These arguments are `variables`, `tmean`, `top` and `nan`.

They work as follows. If you only wanted to select a subset of variables you would use `variables`, as follows:

[8]:
```
df_match = ds.match_points(df, variables = "tas")
```

In some cases, you have monthly point data, but your dataset has daily resolution. In this case you might want a monthly mean output. You can do this using the `tmean` argument:

[9]:
```
df = pd.DataFrame({"lon": [50, 60], "lat": [50, 45], "year":[1850, 1852], "month":[1, 3]}
↪)
df_match = ds.match_points(df, tmean = True)
df_match
```

```
All variables will be used
```

[9]:

	lon	lat	tas	year	month	day
0	50.0	50.0	256.112976	1850	1	16
1	60.0	45.0	271.545959	1852	3	16

This works by applying the dataset `tmean` method to the dataset with the temporal grouping in df. In this case this is the equivalent of running `ds.tmean(["year", "month"])` on the dataset.

When you have a multi-level dataset, but only want the top level, you can set `top=True` in `match_points`. Similarly, if you have a values in the dataset that should be set to missing values, you set them using the `nan` argument.

4.14 Mathematical operations

4.14.1 Creating new variables

Variable creation in nctoolkit can be done using the `assign` method, which works in a similar way to the method available in pandas.

The `assign` method works using lambda functions. Let's say we have a dataset with a variable 'var' and we simply want to add 10 to it and call the new variable 'new'. We would do the following:

```
ds.assign(new = lambda x: x.var + 10)
```

If you are unfamilar with lambda functions, note that the x after lambda signifies that x represents the dataset in whatever comes after ':', which is the actual equation to evaluate. The *x.var* term is *var* from the dataset.

By default assign keeps the original variables in the dataset. However, we may only want the new variable or variables. In that case you can use the drop argument:

```
ds.assign(new = lambda x: x.var+ 10, drop = True)
```

This results in only one variable.

Note that the `assign` method uses kwargs for the lambda functions, so drop can be positioned anywhere. So the following will do the same thing

```
ds.assign(new = lambda x: x.var+ 10, drop = True)
ds.assign(drop = True, new = lambda x: x.var+ 10)
```

At present, `assign` requires that it is written on a single line. So avoid doing something like the following:

```
ds.assign(new = lambda x: x.var+ 10,
drop = True)
```

The *assign* method will evaluate the lambda functions sent to it for each dataset grid cell for each time step. So every part of the lambda function must evaluate to a number. So the following will work:

```
k = 273.15
ds.assign(drop = True, sst_k = lambda x: x.sst + k)
```

However, if you set `k` to a string or anything other than a number it will throw an error. For example, this will throw an error:

```
k = "273.15"
ds.assign(drop = True, sst_k = lambda x: x.sst + k)
```

4.14.2 Applying mathematical functions to dataset variables

As part of your lambda function you can use a number of standard mathematical functions. These all have the same names as those in numpy: `abs`, `floor`, `ceil`, `sqrt`, `exp`, `log10`, `sin`, `cos`, `tan`, `arcsin`, `arccos` and `arctan`.

For example if you wanted to calculate the ceiling of a variable you could do the following:

```
ds.assign(new = lambda x: ceil(x.old))
```

An example of using logs would be the following:

```
ds.assign(new = lambda x: log10(x.old+1))
```

4.14.3 Using spatial statistics

The `assign` method carries out its calculations in each time step, and you can access spatial statistics for each time step when generating new variables. A series of functions are available that have the same names as nctoolkit methods for spatial statistics: `spatial_mean`, `spatial_max`, `spatial_min`, `spatial_sum`, `vertical_mean`, `vertical_max`, `vertical_min`, `vertical_sum`, `zonal_mean`, `zonal_max`, `zonal_min` and `zonal_sum`.

An example of the usefulness of these functions would be if you were working with global temperature data and you wanted to map regions that are warmer than average. You could do this by working out the difference between temperature in one location and the global mean:

```
ds.assign(temp_comp = lambda x: x.temperature - spatial_mean(x.temperature), drop = True)
```

You can also do comparisons. In the above case, we instead might simply want to identify regions that are hotter than the global average. In that case we can simply do this:

```
ds.assign(temp_comp = lambda x: x.temperature > spatial_mean(x.temperature), drop = True)
```

Let's say we wanted to map regions which are 3 degrees hotter than average. We could that as follows:

```
ds.assign(temp_comp = lambda x: x.temperature > spatial_mean(x.temperature + 3), drop =␣
↪True)
```

or like this:

```
ds.assign(temp_comp = lambda x: x.temperature > (spatial_mean(x.temperature)+3), drop =␣
↪True)
```

Logical operators work in the standard Python way. So if we had a dataset with a variable called 'var' and we wanted to find cells with values between 1 and 10, we could do this:

```
ds.assign(one2ten = lambda x: x.var > 1 & x.var < 10)
```

You can process multiple variables at once using `assign`. Variables will be created in the order given, and variables created by the first lambda function can be used by the next one, and so on. The simple example below shows how this works. First we create a var1, which is temperature plus 1. Then var2, which is var1 plus 1. Finally, we calculate the difference between var1 and var2, and this should be 1 everywhere:

```
ds.assign(var1 = lambda x: x.var + 1, var2 = lambda x: x.var1 + 1, diff = lambda x: x.
↪var2 - x.var1)
```

4.14.4 Functions that work with nctoolkit variables

The following functions can be used on nctoolkit variables as part of lambda functions.

Function	Description	Example
`abs`	Absolute value	`abs(x.sst)`
`ceiling`	Ceiling of variable	`ceiling(x.sst -1)`
`cell_area`	Area of grid-cell (m2)	`cell_area(x.var)`
`cos`	Trigonometric cosine of variable	`cos(x.var)`
`day`	Day of the month of the variable	`day(x.var)`
`exp`	Exponential of variable	`exp(x.sst)`
`floor`	Floor of variable	`floor(x.sst + 8.2)`
`hour`	Hour of the day of the variable	`hour(x.var)`
`isnan`	Is variable a missing value/NA?	`isnan(x.var)`
`latitude`	Latitude of the grid cell	`latitude(x.var)`
`level`	Vertical level of variable.	`level(x.var)`
`log`	Natural log of variable	`log10(x.sst + 1)`
`log10`	Base log10 of variable	`log10(x.sst + 1)`
`longitude`	Longitude of the grid cell	`longitude(x.var)`
`month`	Month of the variable	`month(x.var)`
`sin`	Trigonometric sine of variable	`sin(x.var)`
`spatial_max`	Spatial max of variable at time-step	`spatial_max(x.var)`
`spatial_mean`	Spatial mean of variable at time-step	`spatial_mean(x.var)`
`spatial_min`	Spatial min of variable at time-step	`spatial_min(x.var)`
`spatial_sum`	Spatial sum of variable at time-step	`spatial_sum(x.var)`
`sqrt`	Square root of variable	`sqrt(x.sst + 273.15)`
`tan`	Trigonometric tangent of variable	`tan(x.var)`
`timestep`	Time step of variable. Using Python indexing.	`timestep(x.var)`
`year`	Year of the variable	`year(x.var)`
`zonal_max`	Zonal max of variable at time-step	`zonal_max(x.var)`
`zonal_mean`	Zonal mean of variable at time-step	`zonal_mean(x.var)`
`zonal_min`	Zonal min of variable at time-step	`zonal_min(x.var)`
`zonal_sum`	Zonal sum of variable at time-step	`zonal_sum(x.var)`

4.14.5 Simple mathematical operations on variables

If you want to do simple operations like adding or subtracting numbers from the variables in datasets you can use the `add`, `subtract`, `divide` and `multiply` methods. For example if you wanted to add 10 to every variable in a dataset, you would do the following:

```
ds.add(10)
```

If you wanted to multiply everything by 10, you would do this:

```
ds.multiply(10)
```

These methods will also let you use other datasets or netCDF files. So, you could add the values in a dataset data2 to a dataset called data1 as follows:

```
ds1.add(ds2)
```

Please note that this will require that the datasets are structured in a way that the operation makes sense. So each dimension in the datasets will either have to be identical, with the exception of when one dataset has a single value for a dimension. So for example if ds2 above has data covering only 1 timestep, but ds1 has multiple timesteps the data from that single time step will be added to all timesteps in ds1. But if the time steps match, then the data from the first time step in ds2 will be added to the data in the first time step in ds1, and the same will happen with the following time steps.

4.14.6 Simple numerical comparisons

If you want to do something as simple as working out whether the values of the variables in a dataset are greater than zero, you can use the `compare` method. This method accepts a simple comparison formula, which follows Python conventions. For example, if you wanted to figure out if the values in a dataset were greater than zero, you would do the following:

```
ds.compare(">0")
```

If you wanted to know if they were equal to zero you would do this:

```
ds.compare("==0")
```

4.15 Vertical methods

nctoolkit features built in methods for handling files with multiple vertical levels. They work for datasets with fixed vertical levels, for example ocean data with z-levels, as well as files with varying vertical levels such as terrain following coordinates.

The vertical methods available will be illustrated using depth-resolved ocean temperatures from NOAA's World Ocean Atlas for January to a depth of 500 metres. The `vertical_interp` method requires a `levels` argument, which is sea-depth in this case.

[1]:
```
import nctoolkit as nc
nc.deep_clean()
ds = nc.open_thredds("https://data.nodc.noaa.gov/thredds/dodsC/ncei/woa/temperature/A5B7/
↪1.00/woa18_A5B7_t01_01.nc")
```

(continues on next page)

(continued from previous page)

```
ds.subset(variables="t_an")
ds.run()
```

```
nctoolkit is using the latest version of Climate Data Operators version: 2.0.5
```

We can see right awy that there are 57 vertical levels, going as deep as 1500 metres.

[2]:
```
len(ds.levels)
min(ds.levels)
max(ds.levels)
```

[2]:
```
57
```

[2]:
```
0.0
```

[2]:
```
1500.0
```

4.15.1 Calculating simple vertical statistics

If we want to calculate the mean temperature across all vertical levels, we can do the following. Note: that you must specify the fixed arg, which tells nctoolkit whether the vertical levels are consistent in this space. In this case they are. Note: this method will account for cell thickness when calculating the mean.

[3]:
```
ds_mean = ds.copy()
ds_mean.vertical_mean(fixed = True)
ds_mean.plot()
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

[3]:
```
:Overlay
   .Image.I     :Image   [lon,lat]   (t_an)
   .Coastline.I :Feature   [Longitude,Latitude]
```

You can calculate the vertical maximum in a similar way:

[4]:
```
ds_max = ds.copy()
ds_max.vertical_max()
ds_max.plot()
```

[4]:
```
:Overlay
   .Image.I     :Image   [lon,lat]   (t_an)
   .Coastline.I :Feature   [Longitude,Latitude]
```

Likewise, you can calculate the vertical minimum as follows:

[5]:
```
ds_min = ds.copy()
ds_min.vertical_min()
ds_min.plot()
```

[5]:
```
:Overlay
   .Image.I     :Image   [lon,lat]   (t_an)
   .Coastline.I :Feature   [Longitude,Latitude]
```

4.15.2 Vertical interpolation

If you want to carry out vertical interpolation, you can use the `vertical_interp` method. This requires the target levels, and users must specifify whether the vertical levels are fixed in space. In this case, we could interpolate to a single depth of 1000m as follows:

[6]:
```
ds_interp = ds.copy()
ds_interp.vertical_interp(levels = [1000], fixed = True)
ds_interp.plot()
```

[6]:
```
:Overlay
   .Image.I     :Image   [lon,lat]   (t_an)
   .Coastline.I :Feature   [Longitude,Latitude]
```

4.15.3 Selecting the top and bottom levels

You can easily select the top and bottom vertical level using the `top` and `bottom` method.

So, if you wanted to select the sea-surface, you would do the following:

[7]:
```
ds_top = ds.copy()
ds_top.top()
ds_top.plot()
```

[7]:
```
:Overlay
   .Image.I     :Image   [lon,lat]   (t_an)
   .Coastline.I :Feature   [Longitude,Latitude]
```

4.15.4 Additional vertical methods

Other vertical methods available are: `vertical_min`, `vertical_range`, `vertical_cumsum`, and `invert_levels`.

If you are working with ocean data and you want to calculate an integrated water column total, you can use the `vertical_integration` method, which will sum up the water column values accounting for the thickness of each vertical cell.

4.16 Simple arithmetic and comparisons

Basic arithmetic and logical comparisons can be carried out using nctoolkit with the standard Python operators: +, -, *, /, >, <, >=, <=, ==, and !=.

4.16.1 Basic arithmetic using constants and datasets

Often you might want to subtract datasets from each other, or add or subtract a dataset by a constant. The former is potentially made complicated as datasets can take different forms. For example, you might want to subtract a dataset which contains annual means from a dataset that contains monthly values. In this case you want to subtract the annual mean from the relevant month in each year. To deal with this problem, nctoolkit offers the ability to use standard Python operations +, -, * and / to carry out these operations, and in most use-cases it will be able to carry out an appropriate calculation.

Let's illustrate this using a dataset of monthly sea surface temperature from 1850 to the present day. We will start by looking at the first time step:

[1]:
```
import nctoolkit as nc
ds = nc.open_data("sst.mon.mean.nc")
ds.subset(time = 0)
ds.plot()
```

```
nctoolkit is using Climate Data Operators version 2.1.0
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

[1]:
```
:Image   [lon,lat]   (sst)
```

This is temperature in Celsius. However, we might want this in Kelvin. To get this we can simply add 273.15 to the dataset:

[2]:
```
ds+273.15
ds.plot()
```

[2]:
```
:Image   [lon,lat]   (sst)
```

That was easy. And the same is true for adding or subtracting datasets from each other. Let's calculate the temperature in 2010s and see how much warmer it was than in the 1910s:

[3]:
```
ds_2010s = nc.open_data("sst.mon.mean.nc")
ds_2010s.subset(year = range(2010, 2020))
ds_2010s.tmean()
ds_1910s = nc.open_data("sst.mon.mean.nc")
ds_1910s.subset(year = range(1910, 1920))
ds_1910s.tmean()
ds_2010s-ds_1910s
ds_2010s.plot()
```

[3]:
```
:Image   [lon,lat]   (sst)
```

We can see that overall, the world's oceans where much warmer in the 2010s than a century before.

Similarly, if you want to subtract a dataset with only one time step, things will work as expected:

Now, let's think of something slightly more complicated. If I wanted to work out how much warmer or colder each month was than average, how would I do that? You can just use - as above. But in this case nctoolkit we can use the fact that nctoolkit can figure out what it is subtracting from what. Let's start by calculating the mean monthly temperature for all years in our data:

[4]:
```
ds_ave = nc.open_data("sst.mon.mean.nc")
ds_ave.tmean("month")
```

Now, let's now subtract this from our original dataset which has monthly temperature from 1850 to the present day. Once, we have done that we can then calculate a spatial mean to get some idea of long-term trends.

[5]:
```
ds_ave = nc.open_data("sst.mon.mean.nc")
ds_ave.tmean("month")
ds = nc.open_data("sst.mon.mean.nc")
ds-ds_ave
ds.spatial_mean()
ds.plot()
```

```
Subtracting a monthly time series
```

[5]:
```
:DynamicMap   [variable]
   :Curve   [time]   (value)
```

You will notice that nctoolkit has told you it is subtracting a monthly time series. If we were to subtract a time series that only has annual data, we would get a different message:

[6]:
```
ds_ave = nc.open_data("sst.mon.mean.nc")
ds_ave.tmean("year")
ds = nc.open_data("sst.mon.mean.nc")
ds-ds_ave
ds.run()
```

```
Subtracting a yearly time series
```

Note: these methods require consistency between the datasets. For example, in the code below we are subtracting the annual means from the monthly values, but we have removed the year 2000. So running this code will throw an error.

At present, the +, - / and * methods will only be able to handle monthly or yearly datasets on the right-hand side of the operator. That is, it will be able to automatically handle monthly and annual mean data on the right hand-side. An upcoming nctoolkit release will add the ability to automatically handle the ability to handle daily data, so that you can subtract a daily climatology from a dataset with multiple years of daily data.

Importantly, nctoolkit will automatically handle datasets on the right-hand side of the +, -, / or * operators if they have one time step or the same number of time steps as the dataset on the left-hand side. In this case the operation is unambiguous.

nctoolkit also offers verbose methods for these methods, with the following names:

Succinct	Verbose
+	add
-	subtract
/	divide
*	multiply

So, the following would be equivalent:

[7]:
```
ds+273.15
ds.add(273.15)
```

4.16.2 Comparisons

As with simple arithematic, logical comparisons can be done in the standard way in nctoolkit, with the dataset being on the left side of the operator. The right hand side of the operator can be a constant or another dataset of the same structure. The following are available: >, <, ==, >=, <=, !=. We can illustrate this using the temperature dataset used above. If we wanted to calculate where temperature is higher than 10 C, we could do the following:

[8]:
```
ds = nc.open_data("sst.mon.mean.nc")
ds.subset(time = 0)
ds>10
ds.plot()
```

[8]:
```
:Image   [lon,lat]   (sst)
```

In a similar way, we could work out which parts of the world's oceancs were warmer in the 2010s than the 1910s:

[9]:
```
ds_2010s = nc.open_data("sst.mon.mean.nc")
ds_2010s.subset(year = range(2010, 2020))
ds_2010s.tmean()
ds_1910s = nc.open_data("sst.mon.mean.nc")
ds_1910s.subset(year = range(1910, 1920))
ds_1910s.tmean()
ds_2010s>ds_1910s
ds_2010s.plot()
```

[9]:
```
:Image   [lon,lat]   (sst)
```

We can see that the the vast majority was warmer. In fact, we could take it a step further and calculate what fraction was warmer:

[10]:
```
ds_2010s.spatial_mean()
ds_2010s.to_dataframe()
```

[10]:
```
                          time_bnds       sst
time       bnds lon lat
2019-12-01 0    0.0 0.0 2019-12-01  0.965985
           1    0.0 0.0 2019-12-01  0.965985
```

We see that it was over 95%.

4.17 Parallel processing

nctoolkit is written to enable rapid processing and analysis of netCDF files, and this includes the ability to process in parallel. Two methods of parallel processing are available. First is the ability to carry out operations on multi-file datasets in parallel. Second is the ability to define a processing chain in nctoolkit, and then use the multiprocessing or multiprocess package to process files in parallel using that chain. The multiprocessing package is not compatible with nctoolkit internals on macOS, so the multiprocess package should be used instead.

4.17.1 Parallel processing of multi-file datasets

If you have a multi-file dataset, processing the files within it in parallel is easy. All you need to is the following:

```
nc.options(cores = 6)
```

This will tell nctoolkit to process the files in multi-file datasets in parallel and to use 6 cores when doing so. You can, of course, set the number of cores as high as you want. The only thing nctoolkit will do is limit it to the number of cores on your machine.

4.17.2 Parallel processing using multiprocessing or multiprocess

A common task is taking a bunch of files in a folder, doing things to them, and then saving a modified version of each file in a new folder. We want to be able to parallelize that, and we can using the multiprocessing package in the usual way.

But first, we need to change the global settings:

```
import nctoolkit as nc
nc.options(parallel = True)
```

This tells nctoolkit that we are about to do something in parallel. This is critical because of the internal workings of nctoolkit. Behind the scenes nctoolkit is constantly creating and deleting temporary files. It manages this process by creating a safe-list, i.e. a list of files in use that should not be deleted. But if you are running in parallel, you are adding to this list in parallel, and this can cause problems. Telling nctoolkit it will be run in parallel tells it to switch to using a type of list that can be safely added to in parallel.

We can illustrate the use of nctoolkit to post-process multiple files in parallel with a simple chain which will convert files that have temperature in degrees Celsius and then convert them to Kelvin and also save the new outputs as separate files.

First, we would define a function that can take the input file, carry out the necessary processing and then save the output file in a new directory.

In this case, the original file is in a directory called ensemble and we will put it in a new one called new.

```
def process_chain(infile):
    '''
    This function takes a file, converts the temperature to Kelvin and then saves the␣
↪output in a new directory
    '''
```

(continues on next page)

(continued from previous page)

```
    # define the outfile name
    outfile = infile.replace('ensemble', 'new')
    # check if directory for outfile exists and create if not
    if not os.path.exists(os.path.dirname(outfile)):
        os.mkdir(os.path.dirname(outfile))
    ds = nc.open_data(infile)
    # convert to Kelvin
    ds.assign(tos = lambda x: x.sst + 273.15)
    # save the output
    ds.to_nc(outfile)
```

We now want to loop through all of the files in a folder, apply the function to them and then save the results in a new folder called new.

```
# identify files in the ensemble directory
ensemble = nc.create_ensemble("ensemble")
import multiprocessing as mp
import os
# on macOS, use:
#import multiprocess as mp
# create a pool of workers
pool = mp.Pool(3)
# apply the function to each file in the ensemble
for ff in ensemble:
    pool.apply_async(process_chain, [ff])
# close the pool and wait for the work to finish
pool.close()
# wait for the processes to finish
pool.join()
```

The number 3 in this case signifies that 3 cores are to be used.

Please note that if you are working interactively or in a Jupyter notebook, it is best to reset parallel as follows once you have stopped any parallel processing:

```
nc.options(parallel = False)
```

This is because of the effects of manually terminating commands on multiprocessing lists, which nctoolkit uses when in parallel mode. This appears to be due to a book in multiprocessing, which is hard to avoid.

4.18 Examples

This tutorial runs through a number of example work flows.

4.18.1 Global sea surface temperature since 1850

This example analyzes a global sea surface temperature dataset, covering the years since 1850. The data is available from the National Oceanic and Atmospheric Administration (NOAA) here.

We are looking at global sea surface temperature since 1850, so an obvious question is how much the oceans have warmed over this time period. We can use nctoolkit's `spatial_mean` method to calculate this:

Once the file is downloaded, we should set it to ff:

[1]:
```
import nctoolkit as nc
ff = "sst.mon.mean.nc"
```

```
nctoolkit is using Climate Data Operators version 1.9.10
```

[2]:
```
ds = nc.open_data(ff)
ds.spatial_mean()
ds.plot()
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

[2]:
```
:DynamicMap   [variable]
   :Curve   [time]   (value)
```

We can see a clear temperature rise of about 1 degree Celcius. But this is monthly data, so a bit noisy. We can smooth it out by taking an annual mean. In this case we send "year" to `tmean` to tell it to calculate the mean for each year:

[3]:
```
ds = nc.open_data(ff)
ds.tmean(["year"])
ds.spatial_mean()
ds.plot("sst")
```

[3]:
```
:DynamicMap   [variable]
   :Curve   [time]   (value)
```

That is getting better. But again, we possibly want a rolling average. We can use the `rolling_mean` method to calculate the mean over every 20-year period:

[4]:
```
ds = nc.open_data(ff)
ds.tmean(["year"])
ds.spatial_mean()
ds.rolling_mean(20)
ds.plot("sst")
```

[4]:
```
:DynamicMap   [variable]
   :Curve   [time]   (value)
```

We'll finish things off by tweaking this so that we can work out how much temperature has increased since the first 20 years in the time period. For this we can use the `annual_anomaly` method.

[5]:
```
ds = nc.open_data(ff)
ds.annual_anomaly(baseline = [1850, 1869], window = 20)
ds.spatial_mean()
ds.plot("sst")
```

[5]:
```
:DynamicMap   [variable]
   :Curve   [time]   (value)
```

4.18.2 More to come....

4.19 Random Data Hacks

nctoolkit features a number of useful methods to tweak data.

4.19.1 Handling missing values

If you need to set or change missing values, you can use nctoolkit's built-in methods: `as_missing`, `missing_as` and `set_fill`.

Changing an individual value or values within a range to missing values, is easy using `as_missing`. If you wanted to set zeroes to missing values, you would do the following:

```
ds.as_missing(0)
```

In some cases, you might want to set values within a range to missing. In that case, just supply a list to `as_missing`. The following would set all values from -100 to 0 to missing:

```
ds.as_missing([-1000, 0])
```

If you need to change missing values to a constant value, use `missing_as`. The following would change missing values to a constant value of -9999.99:

```
ds.missing_as(-9999.99)
```

Sometimes you might want to change the fill value used in the netCDF file. This can be particularly useful if you are working with muiltiple files with different fill values. You can do this using using `set_fill`:

```
ds.set_fill(-9e38)
```

4.19.2 Shifting time

Sometimes the times in datasets are not quite what we want, and we need some way to adjust time. An example of this is when you are missing a year of data, so want to copy data from the prior year and use it. But first you would need to shift the times in that year forward by a year. You can do this with the `shift` method. This let's you shift time forward by a specified number of hours, days, months or years. You just need to supply hours, days, months or years as an argument. So, if you wanted to shift time backward by one year, you would do the following:

```
ds.shift(years = -1)
```

If you wanted to shift time forward by 12 hours, this would do it:

```
ds.shift(hours = 12)
```

Note: this method allows partial matches to the arguments, so you could use hour, day, month or year just as easily.

4.19.3 Adding cell areas to a dataset

You can add grid cell areas to a dataset as follows:

```
ds.cell_area()
```

By default, this will add the cell area (in square metres) to the dataset. If you want the dataset to only include cell areas you need to set the `join` argument to `False`:

```
ds.cell_area(join = False)
```

Of course, this method will only if it is possible to calculate the areas the grid cells.

4.19.4 Changing the format of the netCDF files in a dataset

Sometimes you will want to change the format of the files in a dataset. You can do this using the `format` method. This let's you set the format, with the following options:

- netCDF = "nc1"
- netCDF version 2 (64-bit offset) = "nc2"/"nc"
- netCDF4 (HDF5) = "nc4"
- netCDF4-classi = "nc4c"
- netCDF version 5 (64-bit data) = "nc5"

So, if you want to set the format to netCDF4, you would do the following:

```
ds.format("nc4")
```

4.19.5 Getting rid of dimensions with only one value

Sometimes you will have a dataset that has a dimension with only one value, and you might want to get rid of that dimension. For example, you might only have one one timestep and keeping it may have no value. Getting rid of that dimension can be done using the `reduce_dims` method. It works as follows:

```
ds.reduce_dims()
```

4.19.6 Removing leap days

If you want to remove a leap day from a dataset, just do the following:

```
ds.drop(month = 2, day = 29)
```

4.19.7 Renaming variables

If you want to rename variables, you can use the *rename* method. Just provide a dictionary where the keys are the original variable names and the values are the new names. So if you wanted to rename a variable x to y, you would do this:

```
ds.rename({"x":"y"})
```

4.20 Global settings

nctoolkit let's you set global settings using options.

4.20.1 Setting the number of cores in use

If you are working with ensembles, you will probably to change the number of cores used for processing multiple files. For example, you can process multiple files in parallel using 6 cores as follows.

```
nc.options(cores = 6)
```

4.20.2 Setting the temporary directory to use

By default nctoolkit uses the OS's temporary directories when it needs to create temporary files. In most cases this is optimal. Most of the time reading and writing to temporary folders is faster. However, in some cases this may not be a good idea because you may not have enough space in the temporary folder. In this case you can change the directory used for saving temporary files as follows:

```
nc.options(temp_dir = "/foo")
```

4.20.3 Turning progress bars on or off

By default, nctoolkit will display a progress bar when it thinks a process will take a long time for a multi-file dataset. If you always want a progress bar to display when calculations are being carried out on multi-file datasets, regardless of their size, you can do the following:

```
nc.options(progress = 'on')
```

If you find the progress bar annoying or distracting, you can just do this:

```
nc.options(progress = 'off')
```

4.20.4 Switching off lazy evaluation

By default evaluation in nctoolkit is lazy, so things are only evaluated when they have to be. If you want things to be evaluated each time a method is used, you can do this:

```
nc.options(lazy = False)
```

4.20.5 Setting global settings using a configuration file

You may want to set some global settings either permanently or on a project level. You can do this by setting up a configruation file. This should be a plain text file called .nctoolkitrc or nctoolkitrc. It should be placed in one of two locations: your working directory or your home directory. When nctoolkit is imported, it will look first in your working directory and then in your home directory for a file called .nctoolkitrc or nctoolkitrc. It will then use the first it finds to change the global settings from the defaults.

The structure of this file is straightforward. For example, if you wanted to set evaluation to lazy and the number of cores used for processing multi-file datasets, you would the following in your configuration file:

lazy : True

cores : 6

The files roughly follow Python dictionary syntax, with the setting and value separate by :. Note that unless the setting is specified in the file, the defaults will be used. If you do not provide a configuration file, nctoolkit will use the default settings.

4.21 Backends

nctoolkit relies on Climate Data Operators (CDO) as its computational backend. This is a high-powered command line tool for manipulating and analyzing climate model data. You can read more about CDO on their website.

nctoolkit is designed as a stand alone package and users will require no understanding of CDO to use it. However, people with knowledge of CDO may want to use the `cdo_command` method to use CDO methods directly.

4.21.1 Using CDO commands

If you want to apply a CDO command in nctoolkit, all you need to do is remove the beginning and end, i.e. 'cdo' and the file names.

So, a typical CDO command looks like this:

```
cdo yearmean infile.nc outfile.nc
```

If we wanted to use this in nctoolkit, we would just do this:

```
ds.cdo_command("yearmean")
```

If the CDO command is an ensemble method that takes multiple files as input and produces one, you will need to specify that it is an ensemble method, as follows:

```
ds.cdo_command("ensmean", ensemble = True)
```

4.21.2 Using NCO commands

nctoolkit also allows you to apply NCO commands to datasets using the `nco_command` method. You just need to remove the two file names from the command you want to apply.

So, the following command:

```
ncks -v kd_490 -d lat,40.0,70.0 -d lon,-20.0,15.0 infile.nc outfile.nc
```

would become:

```
ds.nco_command("ncks -v kd_490 -d lat,40.0,70.0 -d lon,-20.0,15.0")
```

4.22 Toubleshooting

If you get errors running nctoolkit these are most likely caused by problems in the data. In some cases these can be fixed. The tips below will point you help you do this.

4.22.1 Check data types

Under-the-hood, nctoolkit uses Climate Data Operators (CDO) as its computational engine. By default, CDO uses the data type stored in netCDF files. In most cases, this will not cause any problems. However, some times it will. For example, imagine you want to calculate the fraction of time temperature exceeds 30 degrees, but the data is stored as integer format. In nctoolkit, you could calculate this as follows:

```
ds.assign(temperature = lambda x: x.temperature > 30)
ds.tmean()
```

In theory, this is fine. But, if the data is stored as integer format, you will end up either with 0 or 1 in the data. Instead we want to change the numerical precision of the data. We could do this as follows:

```
ds.set_precision("F64")
ds.assign(temperature = lambda x: x.temperature > 30)
ds.tmean()
```

By default, nctoolkit will warn you if a dataset has integer data types when you open a dataset. But if you want to know what data types each variable has just do the following:

```
ds.contents
```

4.22.2 How to carry out general checks on a dataset

There is a built in method in nctoolkit for checking if the format of a dataset is problematic. Just do the following:

```
ds.check()
```

This will carry out a number of checks. First, it will check if there are any variables with integer data types. Second, it will check if the time dimension is stored as an integer data type, which can potentially cause problems. Third, it will check if files are CF-compliant. Lack of CF-compliance could point towards problems with CDO interpreting the contents of the dataset, and thus problems in nctoolkit. Finally, it will check if the variables in a datset have the same horizontal grids.

4.22.3 How to check if a file is corrupt

A common problem with netCDF files is that they can be corrupt. This typically means that parts of the data cannot be accessed. If you want to check if a dataset is corrupt just do the following:

```
ds.is_corrupt()
```

4.22.4 How to fix a dataset with coordinates as variables

Sometimes longitude and latitude will be stored as variables in a netCDF file. Ideally they should be coordinates for nctoolkit to work fully. You can fix this using the `assign_coords` method as follows:

```
ds.assign_coords(lon_name = "lon", lat_name = "lat")
```

where *lon_name* and *lat_name* should be the name of the longitude and latitude variables.

4.23 API Reference

4.23.1 Session options

options(**kwargs)	Define session options.

nctoolkit.options

nctoolkit.**options**(***kwargs*)
Define session options.

Parameters ****kwargs** – Define options using key, value pairs. Set thread_safe = True if hdf5 was built to be thread safe. Set lazy = False if you want methods to evaluate non-lazily Set cores = n, if you want nctoolkit to process the individual files in multi-file datasets in parallel. Note this only applies to multi-file datasets and will not improve performance with single files. Set temp_dir = "/foo" if you want to change the temporary directory used by nctoolkit to save temporary files. Set = "/foo" if you want to change the temporary directory used by nctoolkit to save temporary files. Set progress to "on" or "off" if you always or never want a progress bar to show when multi-file datasets are processed. This defaults to "auto", i.e. nctoolkit will automatically decide whether to show a progress bar based on the size of the ensemble.

Examples

If you wanted to process the files in multi-file datasets in parallel with 6 cores, do the following:

```
>>> import nctoolkit as nc
>>> nc.options(cores = 6)
```

If you want to set evaluation to always be lazy do the following:

```
>>> nc.options(lazy = True)
```

If you want nctoolkit to store temporary files in a specific directory, do this:

```
>>> nc.options(temp_dir = "/foo")
```

4.23.2 Opening/copying data

open_data([x, checks])	Read netCDF data as a Dataset object
open_url([x, ftp_details, wait, file_stop])	Read netCDF data from a url as a DataSet object
open_thredds([x, wait, checks])	Read thredds data as a Dataset object
open_geotiff([x])	Open a geotiff and convert to a Dataset This requires rioxarray to be installed.
from_xarray(ds)	Convert an xarray dataset to an nctoolkit dataset This will first save the xarray dataset as a temporary netCDF file.
DataSet.copy(self)	Make a deep copy of an DataSet object.

nctoolkit.open_data

nctoolkit.**open_data**(*x=[]*, *checks=True*, ***kwargs*)
Read netCDF data as a Dataset object

Parameters

- **x** (*str or list*) – A string or list of netCDF files or a single url. The function will check the files exist. If x is not a list, but an iterable it will be converted to a list. If a *.nc style wildcard is supplied, open_data will use all files available. By default an empty dataset is created, ie. using open_data() will create an empty dataset that can then be expanded using append.
- **checks** (*boolean*) – Do you want basic checks to ensure cdo can read files? Default to True. Setting to False can result in a minor speed up.
- ****kwargs** (*kwargs*) – Optional arguments for internal use by open_thredds and open_url.

Returns open_data

Return type nctoolkit.DataSet

Examples

If you want to open a single file as a dataset, do the following:

```
>>> import nctoolkit as nc
>>> ds = nc.open_data("example.nc")
```

If you want to open a list of files as a multi-file dataset, you would do something like this:

```
>>> import nctoolkit as nc
>>> ds = nc.open_data(["file1.nc", "file2.nc", "file3.nc"])
```

If you wanted to open all files in a directory "data" as a multi-file dataset, you can use a wildcard:

```
>>> import nctoolkit as nc
>>> ds = nc.open_data("data/*.nc")
```

nctoolkit.open_url

nctoolkit.**open_url**(*x=None*, *ftp_details=None*, *wait=None*, *file_stop=None*)
Read netCDF data from a url as a DataSet object

Parameters

- **x** (*str*) – A string with a url. Prior to processing data will be downloaded to a temp folder.
- **ftp_details** (*dict*) – A dictionary giving the user name and password combination for ftp downloads: {"user":user, "password":pass}
- **wait** (*int*) – Time to wait, in seconds, for data to download. A minimum of 3 attempts will be made to download the data.
- **file_stop** (*int*) – Time limit, in minutes, for individual attempts at downloading data. This is useful to get around download freezes.

Returns open_url

Return type nctoolkit.DataSet

Examples

If you want to open a file available over a url do the following:

```
>>> import nctoolkit as nc
>>> ds = nc.open_url("htttp:://foo.nc")
```

This will download the file as a temporary folder for use in the dataset.

nctoolkit.open_thredds

nctoolkit.**open_thredds**(*x=None*, *wait=None*, *checks=False*)

Read thredds data as a Dataset object

Parameters

- **x** (*str or list*) – A string or list of thredds urls, which must end with .nc.
- **checks** (*boolean*) – Do you want to check if data is available over thredds?
- **wait** (*int*) – Time to wait for thredds server to be checked. Limitless if not supplied.

Returns open_thredds

Return type nctoolkit.DataSet

Examples

If you want to open a file available over thredds or opendap, do the following:

```
>>> import nctoolkit as nc
>>> ds = nc.open_thredds("htttp:://foo.nc")
```

nctoolkit.open_geotiff

nctoolkit.**open_geotiff**(*x=[]*)

Open a geotiff and convert to a Dataset This requires rioxarray to be installed.

Parameters **x** (*str or list*) – A string or list of geotiff files or a single url. This requires rioxarray to be installed.

Returns open_geotiff

Return type nctoolkit.DataSet

nctoolkit.from_xarray

nctoolkit.**from_xarray**(*ds*)

Convert an xarray dataset to an nctoolkit dataset This will first save the xarray dataset as a temporary netCDF file.

Parameters **ds** (*xarray dataset*) – xarray dataset you want to convert to nctoolkit DataSet.

Returns **from_xarray**

Return type nctoolkit.DataSet

nctoolkit.DataSet.copy

DataSet.**copy**(*self*)

Make a deep copy of an DataSet object. Note: This will not make disk copies of the temporary files underlying datasets, so it will be disk-space efficient.

Returns **copy**

Return type nctoolkit DataSet

4.23.3 Merging or analyzing multiple datasets

merge(*datasets[, match])	Merge datasets
cor_time([x, y])	Calculate the temporal correlation coefficient between two datasets This will calculate the temporal correlation coefficient, for each time step, between two datasets.
cor_space([x, y])	Calculate the spatial correlation coefficient between two datasets This will calculate the spatial correlation coefficient, for each time step, between two datasets.

nctoolkit.merge

nctoolkit.**merge**(**datasets, match=['day', 'year', 'month']*)

Merge datasets

Parameters

- **datasets** (*kwargs*) – Datasets to merge.
- **match** (*list*) – Temporal matching criteria. This is a list which must be made up of a subset of day, year, month. This checks that the datasets have compatible times. For example, if you want to ensure the datasets have the same years, then use match = [“year”].

nctoolkit.cor_time

nctoolkit.**cor_time**(*x=None*, *y=None*)

Calculate the temporal correlation coefficient between two datasets This will calculate the temporal correlation coefficient, for each time step, between two datasets. The datasets must either have the same variables or only have one variable.

Parameters

- **x** (*dataset*) – First dataset to use
- **y** (*dataset*) – Second dataset to use

Returns **cor_time** – A dataset containing the temporal correlation coefficient for each grid cell

Return type nctoolkit Dataset

nctoolkit.cor_space

nctoolkit.**cor_space**(*x=None*, *y=None*)

Calculate the spatial correlation coefficient between two datasets This will calculate the spatial correlation coefficient, for each time step, between two datasets. The datasets must either have the same variables or only have one variable.

Returns **cor_space** – A dataset containing the correlation coefficient for each time step.

Return type nctoolkit Dataset

Parameters

- **x** (*dataset*) – First dataset to use
- **y** (*dataset*) – Second dataset to use

4.23.4 Adding and removing files to a dataset

DataSet.append(self[, x])	append: Add new file(s) to a dataset.
DataSet.remove(self[, x])	remove: Remove file(s) from a dataset

nctoolkit.DataSet.append

DataSet.**append**(*self*, *x=None*)

append: Add new file(s) to a dataset.

Parameters **x** (*str or list*) – File path(s) to add to the dataset

Examples

If you want to add a dataset ds2 to another dataset ds1, do the following:

```
>>> ds1.append(ds2)
```

If you want to add a new file to a dataset, do this:

```
>>> ds.append("infile.nc")
```

nctoolkit.DataSet.remove

DataSet.**remove**(*self*, *x=None*)

remove: Remove file(s) from a dataset

Parameters **x** (*str or list*) – File path(s) to remove from a dataset

Examples

If you want to remove a file from a dataset do the following:

```
>>> ds.remove("infile.nc")
```

4.23.5 Accessing attributes

DataSet.variables	List variables contained in a dataset
DataSet.contents	Detailed list of variables contained in a dataset.
DataSet.times	List times contained in a dataset
DataSet.years	List years contained in a dataset
DataSet.months	List months contained in a dataset
DataSet.levels	List levels contained in a dataset
DataSet.size	The size of an object This will print the number of files, total size, and smallest and largest files in an DataSet object.
DataSet.current	The current file or files in the DataSet object
DataSet.history	The history of operations on the DataSet
DataSet.start	The starting file or files of the DataSet object
DataSet.calendar	List calendars of dataset files
DataSet.ncformat	List formats of files contained in a dataset

nctoolkit.DataSet.variables

property DataSet.**variables**
List variables contained in a dataset

Returns List of variables

Return type list

nctoolkit.DataSet.contents

property DataSet.**contents**
Detailed list of variables contained in a dataset. This will only display the variables in the first file of an ensemble.

Returns A pandas dataframe containing the variables, number of points, number of levels, long name, units, and data type of each variable in the dataset.

Return type pandas.DataFrame

nctoolkit.DataSet.times

property DataSet.**times**
List times contained in a dataset

Returns List of times in the dataset

Return type list

nctoolkit.DataSet.years

property DataSet.**years**
List years contained in a dataset

Returns List of years in the dataset

Return type list

nctoolkit.DataSet.months

property DataSet.**months**
List months contained in a dataset

Returns List of months in the dataset

Return type list

nctoolkit.DataSet.levels

property DataSet.**levels**
List levels contained in a dataset

Returns List of levels in the dataset

Return type list

nctoolkit.DataSet.size

property DataSet.**size**
The size of an object This will print the number of files, total size, and smallest and largest files in an DataSet object.

Returns **size** – A dictionary containing the number of files, total size, and smallest and

Return type dict

nctoolkit.DataSet.current

property DataSet.**current**
The current file or files in the DataSet object

Returns A list of the current files in the DataSet object

Return type list

nctoolkit.DataSet.history

property DataSet.**history**
The history of operations on the DataSet

Returns A list of the operations performed on the DataSet

Return type list

nctoolkit.DataSet.start

property DataSet.**start**
The starting file or files of the DataSet object

Returns A list of the starting files in the DataSet object

Return type list

nctoolkit.DataSet.calendar

property DataSet.**calendar**
List calendars of dataset files

Returns List of calendars in the dataset files

Return type list

nctoolkit.DataSet.ncformat

property DataSet.**ncformat**
List formats of files contained in a dataset

Returns List of netCDF formats in the dataset

Return type list

4.23.6 Plotting

DataSet.plot(self[, vars, autoscale, out, coast])	plot: Automatically plot a dataset.

nctoolkit.DataSet.plot

DataSet.**plot**(*self*, *vars=None*, *autoscale=True*, *out=None*, *coast=None*, ***kwargs*)
plot: Automatically plot a dataset.

Parameters

- **vars** (*str , list*) – A string or list of the variables to plot
- **autoscale** (*bool*) – Set to True if you want the colorbar to be scaled to the min/max of the data. Default is True
- **vars** – A string or list of the variables to plot
- **out** (*str*) – Name of output file if you want to save as html. Defaults to None.
- **coast** (*bool*) – Set to True if you want a coastline to show up on spatial map. Default is True if cartopy is installed and working. Otherwise it is False.
- ****kwargs** (*Optional args to be sent to hvplot*) –

Returns

Return type If out is None, returns a hvplot object. Otherwise, saves a html file to the location specified by out.

Examples

If you want to plot all data in a dataset, do the following:

```
>>> ds.plot()
```

If you only want to plot a single variable, do the following. Note, this is often faster if you have a large dataset.

```
>>> ds.plot("var_of_choice")
```

4.23.7 Variable modification

DataSet.assign(self[, drop])	assign: Create new variables using mathematical operations on existing variables.
DataSet.rename(self[, newnames])	rename: Rename variables in a dataset
DataSet.as_missing(self[, value])	Change a range or individual value to missing.
DataSet.missing_as(self[, value])	Convert missing values to a constant
DataSet.set_fill(self[, value])	Set the fill value
DataSet.sum_all(self[, drop, new_name])	sum_all: Calculate the sum of all variables for each time step

nctoolkit.DataSet.assign

DataSet.**assign**(*self*, *drop=False*, ***kwargs*)

assign: Create new variables using mathematical operations on existing variables.

Existing columns that are re-assigned will be overwritten. This method operators in a similar fashion to the pandas assign method.

Parameters

- **drop** (*bool*) – Set to True if you want existing variables to be removed once the new ones have been created. Defaults to False.
- ****kwargs** (*dict of {str: callable}*) – New variable names are keywords. All terms in the equation given by the lambda function should evaluate to a numeric. New variables are calculated for each grid cell and time step.

Notes

Operations are carried out in the order give. So if a new variable is created in the first argument, it can then be used in following arguments.

Examples

Temperature could be converted from Celsius to Kelvin by:

```
>>> ds.assign(temperature_K=lambda x: x.temperature + 273.15)
```

Temperatures higher than the spatial average temperature could be found by:

```
>>> ds.assign(hot=lambda x: x.temperature > spatial_mean(x.temperature))
```

nctoolkit.DataSet.rename

DataSet.**rename**(*self*, *newnames=None*, ***kwargs*)

rename: Rename variables in a dataset

Parameters

- **newnames** (*dict*) – Dictionary with key-value pairs being original and new variable names
- **kwargs** (*) – Alternative method for renaming

Examples

If you want to rename a variable x to y, do the following:

```
>>> ds.rename({"x":"y"})
```

Alternatively, you could do the following:

```
>>> ds.rename(x="y")
```

nctoolkit.DataSet.as_missing

DataSet.**as_missing**(*self*, *value=None*)

Change a range or individual value to missing.

Parameters **value** (*2 variable list or int/float*) – If int/float is provided, the missing value will be set to that. If a list is provided, values between the two values (inclusive) of the list are set to missing.

Examples

Set all zeros to missing: >>> ds.as_missing(0)

Set all values between 0 and 1 to missing: >>> ds.as_missing([0, 1])

nctoolkit.DataSet.missing_as

DataSet.**missing_as**(*self*, *value=None*)

Convert missing values to a constant

Parameters **value** (*int/float*) – If int/float is provided, the missing value will be converted to that.

Examples

Convert all missing values to 0: >>> ds.missing_as(0)

nctoolkit.DataSet.set_fill

DataSet.**set_fill**(*self*, *value=None*)

Set the fill value

Parameters **value** (*int/float*) – The fill value to set.

Examples

Set the fill value to -9999 >>> ds.set_fill(-9999)

nctoolkit.DataSet.sum_all

DataSet.**sum_all**(*self*, *drop=True*, *new_name=None*)

sum_all: Calculate the sum of all variables for each time step

Parameters

- **drop** (*boolean*) – Do you want to keep variables?
- **new_name** (*string*) – If you want to name the output of sum_all to a specific name

Examples

```
>>> ds.sum_all()
Setting the name of the output to combined
>>> ds.sum_all(new_name = "combined")
```

4.23.8 netCDF file attribute modification

DataSet.set_longnames(self[, name_dict])	Set the long names of variables
DataSet.set_units(self[, unit_dict])	Set the units for variables

nctoolkit.DataSet.set_longnames

DataSet.**set_longnames**(*self*, *name_dict=None*, ***kwargs*)

Set the long names of variables

Parameters

- **name_dict** (*dict*) – Dictionary with key, value pairs representing the variable names and their long names
- **kwargs** (*) – Alternative method for setting units

Examples

Set the long name of the variable tas to "Temperature" using the dictionary approach: >>> ds.set_longnames(name_dict={"tas": "Temperature"}) Alternatively, use the kwargs approach: >>> ds.set_longnames(tas="Temperature")

nctoolkit.DataSet.set_units

DataSet.**set_units**(*self*, *unit_dict=None*, ***kwargs*)

Set the units for variables

Parameters

- **unit_dict** (*dict*) – A dictionary where the key-value pairs are the variables and new units respectively.
- **kwargs** (*) – Alternative method for setting units using direct assignment

Examples

Set the units for a variable called 'tas' to 'K': >>> ds.set_units({'tas': 'K'})

Set the units for a variable called 'tas' to 'K' using kwargs: >>> ds.set_units(tas='K')

4.23.9 Vertical/level methods

DataSet.top(self)	top: Extract the top/surface level from a dataset
DataSet.bottom(self)	bottom: Extract the bottom level from a dataset
DataSet.vertical_interp(self[, levels, ...])	vertical_interp: Verticaly interpolate a dataset based on given vertical levels
DataSet.vertical_mean(self[, thickness, ...])	vertical_mean: Calculate the depth-averaged mean for each variable.
DataSet.vertical_min(self)	vertical_min: Calculate the vertical minimum of variable values.

continues on next page

Table 9 – continued from previous page

DataSet.vertical_max(self)	vertical_max: Calculate the vertical maximum of variable values.
DataSet.vertical_range(self)	vertical_range: Calculate the vertical range of variable values.
DataSet.vertical_sum(self)	vertical_sum: Calculate the vertical sum of variable values.
DataSet.vertical_integration(self[, ...])	vertical_integration: Calculate the vertically integrated sum over the water column.
DataSet.vertical_cumsum(self)	vertical_cumsum: Calculate the vertical sum of variable values.
DataSet.invert_levels(self)	Invert the levels of 3D variables.
DataSet.bottom_mask(self)	bottom_mask: Create a mask identifying the deepest cell without missing values..

nctoolkit.DataSet.top

DataSet.**top**(*self*)

top: Extract the top/surface level from a dataset

This extracts the first vertical level from each file in a dataset.

This method is most useful for things like oceanic data, where this method will extract the sea surface.

You may need to double check that the first vertical level is the surface, as this is not always the case.

Examples

If you wanted to extract the top vertical level of a dataset, do the following:

```
>>> ds.top()
```

nctoolkit.DataSet.bottom

DataSet.**bottom**(*self*)

bottom: Extract the bottom level from a dataset

This extracts the bottom level from each netCDF file. Please note that for ensembles, it uses the first file to derive the index of the bottom level.

You may need to double check that the bottom vertical level is the sea 'bottom' etc., as this is not always the case.

Use bottom_mask for files when the bottom cell in netCDF files do not represent the actual bottom.

Examples

If you wanted to extract the bottom vertical level of a dataset, do the following:

```
>>> ds.bottom()
```

This method is most useful for things like oceanic model data, where the bottom cell corresponds to the bottom of the ocean.

nctoolkit.DataSet.vertical_interp

DataSet.**vertical_interp**(*self*, *levels=None*, *fixed=None*, *thickness=None*, *depths=None*, *surface=None*)

vertical_interp: Verticaly interpolate a dataset based on given vertical levels

Vertical interpolation is calculated for each time step and grid cell

---

Note: This requires consistent vertical levels in space. For the likes of sigma-coordinates, please use to_zlevels.

Parameters

- **levels** (*list, int or str*) – list of vertical levels, for example depths for an ocean model, to vertically interpolate to. These must be floats or ints.
- **fixed** (*bool*) – Define whether the vertical levels are the same in all spatial locations. Set to True if they are, e.g. you have z-levels. If you have the likes of sigma-coordinates, set this to False.
- **thickness** (*str or Dataset*) – This or depths must be supplied if fixed is False, otherwise vertical thickness/depth cannot be known. Option argument when vertical levels vary in space. One of: a variable, in the dataset, which contains the variable thicknesses; a .nc file which contains the thicknesses; or a Dataset that contains the thicknesses. Note: the .nc file or Dataset must only contain one variable. Thickness should be in metres. Vertical interpolation will take the value from the mid-point of the level.
- **depths** (*str or Dataset*) – This or thickness must be supplied if fixed is False, otherwise vertical thickness/depth cannot be known. Option argument when vertical levels vary in space. One of: a variable, in the dataset, which contains the variable depths; a .nc file which contains the depths; or a Dataset that contains the depths. Note: the .nc file or Dataset must only contain one variable. Depths should be in metres, and be the mid-point of the level.
- **surface** (*str*) – If thickness is supplied you must also supply this to identify whether the top or bottom of the level is the surface, i.e. the lowest level. This must be one of 'top' or 'bottom'.

Examples

If you wanted to vertically interpolate a dataset with spatially consistent vertical levels to 5 and 10 metres, you would do the following:

```
>>> ds.vertical_interp(levels = [5,10], fixed = True)
```

This method is most useful for things like oceanic data, where you need to interpolate to certain depth levels. It will require that vertical levels are the same in every grid cell.

nctoolkit.DataSet.vertical_mean

DataSet.**vertical_mean**(*self*, *thickness=None*, *depth_range=None*, *fixed=None*)

vertical_mean: Calculate the depth-averaged mean for each variable.

This is calculated for each time step and grid cell.

thickness: str or Dataset This must be supplied when vertical levels vary in space, i.e. fixed=False. One of: a variable, in the dataset, which contains the variable thicknesses; a .nc file which contains the thicknesses; or a Dataset that contains the thicknesses. Note: the .nc file or Dataset must only contain one variable.

depth_range: list Only use when vertical levels vary in space Set a depth range if desired. Should be of the form [min_depth, max_depth].

fixed [bool] Define whether the vertical levels are the same in all spatial locations. Set to True if they are, e.g. you have z-levels. If you have the likes of sigma-coordinates, set this to True.

Examples

If you wanted to vertical mean of every variable in a dataset with consistent vertical levels, you would do this:

```
>>> ds.vertical_mean(fixed = True)
```

This method will calculate the vertical mean weighted by the thickness of each cell. Note that if cell thickness cannot be derived it will just average the values in each vertical cell.

nctoolkit.DataSet.vertical_min

DataSet.**vertical_min**(*self*)

vertical_min: Calculate the vertical minimum of variable values.

This is calculated for each time step and grid cell.

Examples

If you wanted to vertical minimum of every variable in a dataset, you would do this:

```
>>> ds.vertical_min()
```

nctoolkit.DataSet.vertical_max

DataSet.**vertical_max**(*self*)

vertical_max: Calculate the vertical maximum of variable values.

This is calculated for each time step and grid cell.

Examples

If you wanted to vertical maximum of every variable in a dataset, you would do this:

```
>>> ds.vertical_max()
```

nctoolkit.DataSet.vertical_range

DataSet.**vertical_range**(*self*)
vertical_range: Calculate the vertical range of variable values.

This is calculated for each time step and grid cell.

Examples

If you wanted to range of values across all vertical levels of every variable in a dataset, you would do this:

```
>>> ds.vertical_range()
```

nctoolkit.DataSet.vertical_sum

DataSet.**vertical_sum**(*self*)
vertical_sum: Calculate the vertical sum of variable values.

This is calculated for each time step and grid cell.

Examples

If you wanted to sum of values across all vertical levels of every variable in a dataset, you would do this:

```
>>> ds.vertical_sum()
```

nctoolkit.DataSet.vertical_integration

DataSet.**vertical_integration**(*self*, *thickness=None*, *depth_range=None*, *fixed=None*)
vertical_integration: Calculate the vertically integrated sum over the water column.

This calculates the sum of the variable multiplied by the cell thickness

Parameters

- **thickness** (*str or Dataset*) – This must be supplied when vertical levels vary spatially. One of: a variable, in the dataset, which contains the variable thicknesses; a .nc file which contains the thicknesses; or a Dataset that contains the thicknesses. Note: the .nc file or DataSet must only contain one variable.
- **depth_range** (*list*) – Set a depth range if desired. Should be of the form [min_depth, max_depth].
- **fixed** (*bool*) – Define whether the vertical levels are the same in all spatial locations. Set to True if they are, e.g. you have z-levels. If you have the likes of sigma-coordinates, set this to True.

Examples

If you wanted to integrate values across all vertical levels of every variable in a dataset that has spatially fixed vertical levels, you would do this:

```
>>> ds.vertical_integration(fixed = True)
```

nctoolkit.DataSet.vertical_cumsum

DataSet.**vertical_cumsum**(*self*)

vertical_cumsum: Calculate the vertical sum of variable values.

This is calculated for each time step and grid cell.

Examples

If you wanted to calculate the cumulative sum of values across all vertical levels of every variable in a dataset, you would do this:

```
>>> ds.vertical_sum()
```

The cumulative sum will be calculated from the first to the last vertical level. For example, in oceanic data it would start at the sea surface.

nctoolkit.DataSet.invert_levels

DataSet.**invert_levels**(*self*)

Invert the levels of 3D variables.

This is calculated for each time step and grid cell.

Examples

If you wanted to invert the vertical levels, you would do this:

```
>>> ds.invert_levels()
```

nctoolkit.DataSet.bottom_mask

DataSet.**bottom_mask**(*self*)

bottom_mask: Create a mask identifying the deepest cell without missing values..

This converts a dataset to a mask identifying which cell represents the bottom, for example the seabed. 1 identifies the deepest cell with non-missing values. Everything else is 0, or missing.

Note: This will only work for single file datasets. The method will modify the dataset in place, so make a copy if you want to keep the original.

Examples

If you wanted to create a mask identifying the bottom, you would do this:

```
>>> ds.bottom_mask()
```

4.23.10 Rolling methods

DataSet.rolling_mean(self[, window, align])	rolling_mean: Calculate a rolling mean based on a window
DataSet.rolling_min(self[, window, align])	rolling_min: Calculate a rolling minimum based on a window
DataSet.rolling_max(self[, window, align])	rolling_max: Calculate a rolling maximum based on a window
DataSet.rolling_sum(self[, window, align])	rolling_sum: Calculate a rolling sum based on a window
DataSet.rolling_range(self[, window, align])	rolling_range: Calculate a rolling range based on a window
DataSet.rolling_stdev(self[, window, align])	rolling_stdev: Calculate a rolling standard deviation based on a window
DataSet.rolling_var(self[, window, align])	rolling_var: Calculate a rolling variance based on a window

nctoolkit.DataSet.rolling_mean

DataSet.**rolling_mean**(*self*, *window=None*, *align='right'*)
rolling_mean: Calculate a rolling mean based on a window

Parameters

- **window** (*int*) – The size of the window for the calculation of the rolling mean
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are “left”, “centre” and “right”

Examples

If you wanted to calculate a rolling mean with the mean calculated over every 10 time steps, do the following:

```
>>> ds.rolling_mean(10)
```

nctoolkit.DataSet.rolling_min

DataSet.**rolling_min**(*self*, *window=None*, *align='right'*)
rolling_min: Calculate a rolling minimum based on a window

Parameters

- **window** (*int*) – The size of the window for the calculation of the rolling minimum
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are “left”, “centre” and “right”

Examples

If you wanted to calculate a rolling minimum with the minimum calculated over every 10 time steps, do the following:

```
>>> ds.rolling_min(10)
```

nctoolkit.DataSet.rolling_max

DataSet.**rolling_max**(*self*, *window=None*, *align='right'*)

rolling_max: Calculate a rolling maximum based on a window

Parameters

- **window** (*int*) – The size of the window for the calculation of the rolling maximum
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you wanted to calculate a rolling maximum with the maximum calculated over every 10 time steps, do the following:

```
>>> ds.rolling_max(10)
```

nctoolkit.DataSet.rolling_sum

DataSet.**rolling_sum**(*self*, *window=None*, *align='right'*)

rolling_sum: Calculate a rolling sum based on a window

Parameters

- **window** (*int*) – The size of the window for the calculation of the rolling sum
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you wanted to calculate a rolling sum with the sum calculated over every 10 time steps, do the following:

```
>>> ds.rolling_sum(10)
```

nctoolkit.DataSet.rolling_range

DataSet.**rolling_range**(*self*, *window=None*, *align='right'*)
rolling_range: Calculate a rolling range based on a window

Parameters

- **window** (*int*) – The size of the window for the calculation of the rolling range
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you wanted to calculate a rolling range with the range calculated over every 10 time steps, do the following:

```
>>> ds.rolling_range(10)
```

nctoolkit.DataSet.rolling_stdev

DataSet.**rolling_stdev**(*self*, *window=None*, *align='right'*)
rolling_stdev: Calculate a rolling standard deviation based on a window

Parameters

- **window** (*int*) – The size of the window for the calculation of the rolling standard deviation
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you wanted to calculate a rolling standard deviation with the standard deviation calculated over every 10 time steps, do the following:

```
>>> ds.rolling_stdev(10)
```

nctoolkit.DataSet.rolling_var

DataSet.**rolling_var**(*self*, *window=None*, *align='right'*)
rolling_var: Calculate a rolling variance based on a window

Parameters

- **window** (*int*) – The size of the window for the calculation of the rolling variance
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you wanted to calculate a rolling variance with the variance calculated over every 10 time steps, do the following:

```
>>> ds.rolling_var(10)
```

4.23.11 Evaluation setting

DataSet.run(self)	Run all stored commands in a dataset

nctoolkit.DataSet.run

DataSet.**run**(*self*)
Run all stored commands in a dataset

Examples

If evaluation is lazy and you need to evaluate commands on a dataset, do the following:

```
>>> ds.run()
```

4.23.12 Cleaning functions

—

4.23.13 Ensemble creation

create_ensemble([path, recursive])	create_ensemble: Generate an ensemble of files from a directory.

nctoolkit.create_ensemble

nctoolkit.**create_ensemble**(*path=''*, *recursive=True*)
create_ensemble: Generate an ensemble of files from a directory.

Parameters

- **path** (*str*) – The directory to search for netCDF files
- **recursive** (*boolean*) – True/False depending on whether you want to search the path recursively. Defaults to True.

Returns files

Return type list of files

Examples

If you wanted to recursively find all netCDF files available in a directory "data", you would do this:

```
>>> import nctoolkit as nc
>>> nc.create_ensemble("data")
```

If you wanted to find the files in that directory and ignore subdirectories, you would instead do this:

```
>>> nc.create_ensemble("data", recursive = False)
```

4.23.14 Arithemetic methods

DataSet.abs(self)	abs: Method to get the absolute value of variables
DataSet.add(self[, x, var])	add: Add to a dataset
DataSet.assign(self[, drop])	assign: Create new variables using mathematical operations on existing variables.
DataSet.exp(self)	exp: Method to get the exponential of variables
DataSet.log(self)	log: Method to get the natural log, ln, of variables
DataSet.log10(self)	log10: Method to get the base 10 log, log10, of variables
DataSet.multiply(self[, x, var])	multiply: Multiply a dataset.
DataSet.power(self[, x])	power: Powers of variables in dataset
DataSet.sqrt(self)	sqrt: Method to get the square root of variables
DataSet.square(self)	square: Method to get the square of variables
DataSet.subtract(self[, x, var])	subtract: Subtract from a dataset.
DataSet.divide(self[, x, var])	divide: Divide the data.

nctoolkit.DataSet.abs

DataSet.**abs**(*self*)

abs: Method to get the absolute value of variables

Examples

If you wanted to get the absolute value of each variable, you just need do this:

```
>>> ds.abs()
```

nctoolkit.DataSet.add

DataSet.**add**(*self*, *x=None*, *var=None*)

add: Add to a dataset

This will add a constant, another dataset or a netCDF file to the dataset. nctoolkit will automatically determine the appropriate comparison required.

Parameters

- **x** (*int, float, DataSet or netCDF file*) – An int, float, single file dataset or netCDF file to add to the dataset. If a dataset or netCDF file is supplied, this must have only one variable, unless var is provided. The grids must be the same.

- **var** (*str*) – A variable in the x to use for the operation

Examples

If you wanted to add 10 to all variables in a dataset, you would do the following:

```
>>> ds.add(10)
```

Or, you could use standard python addition syntax:

```
>>> ds + 10
```

To add the values in a dataset ds2 from a dataset ds1, you would do the following:

```
>>> ds1.add(ds2)
```

Or, you could use standard python addition syntax:

```
>>> ds1 + ds2
```

Grids in the datasets must match. Addition will occur in matching timesteps in ds1 and ds2. If there is only 1 timestep in ds2, then the data from that timestep will be added to the data in all ds1 time steps.

Adding the data from another netCDF file will work in the same way:

```
>>> ds1.add("example.nc")
```

nctoolkit.DataSet.exp

DataSet.**exp**(*self*)
exp: Method to get the exponential of variables

Examples

If you wanted to calculate the exponential of a variable, you just need to do this:

```
>>> ds.exp(0.5)
```

nctoolkit.DataSet.log

DataSet.**log**(*self*)
log: Method to get the natural log, ln, of variables

Examples

If you wanted to calculate the natural log of each variable, you just need to do this:

```
>>> ds.log()
```

nctoolkit.DataSet.log10

DataSet.**log10**(*self*)
log10: Method to get the base 10 log, log10, of variables

Examples

If you wanted to calculate the base 10 log of each variable, you just need to do this:

```
>>> ds.log10()
```

nctoolkit.DataSet.multiply

DataSet.**multiply**(*self*, *x=None*, *var=None*)
multiply: Multiply a dataset.

This will multiply a dataset by a constant, another dataset or a netCDF file.

Parameters

- **x** (*int, float, DataSet or netCDF file*) – An int, float, single file dataset or netCDF file to multiply the dataset by. If multiplying by a dataset or single file there must only be a single variable in it, unless var is supplied. The grids must be the same.
- **var** (*str*) – A variable in the x to multiply the dataset by

Examples

If you wanted to multiply variables in a dataset by 10, you would do the following:

```
>>> ds.multiply(10)
```

Or, you could use standard python multiplication syntax:

```
>>> ds * 10
```

To multiply the values in a dataset by the values of variables in dataset ds2, you would do the following:

```
>>> ds1.multiply(ds2)
```

Or, you could use standard python multiplication syntax:

```
>>> ds1 * ds2
```

Grids in the datasets must match. Multiplication will occur in matching timesteps in ds1 and ds2. If there is only 1 timestep in ds2, then the data from that timestep in ds2 will multiply the data in all timesteps in ds1.

Multiplying a dataset by the data from another netCDF file will work in the same way:

```
>>> ds.multiply("example.nc")
```

nctoolkit.DataSet.power

DataSet.**power**(*self*, *x=None*)
power: Powers of variables in dataset

Parameters **x** (*int, float*) – An int or float to take the variables to the power of

Examples

If you wanted to take each variable to the power of 0.5 you would do this:

```
>>> ds.power(0.5)
```

nctoolkit.DataSet.sqrt

DataSet.**sqrt**(*self*)
sqrt: Method to get the square root of variables

Examples

If you wanted to calculate the square root of each variable, you just need to do this:

```
>>> ds.sqrt()
```

nctoolkit.DataSet.square

DataSet.**square**(*self*)
square: Method to get the square of variables

Examples

If you wanted to calculate the square of each variable, you just need to do this:

```
>>> ds.square()
```

nctoolkit.DataSet.subtract

DataSet.**subtract**(*self*, *x=None*, *var=None*)
subtract: Subtract from a dataset.

This will subtract a constant, another dataset or a netCDF file from the dataset.

Parameters

- **x** (*int, float, DataSet or netCDF file*) – An int, float, single file dataset or netCDF file to subtract from the dataset. If a dataset or netCDF is supplied this must only have one variable, unless var is provided. The grids must be the same.

- **var** (*str*) – A variable in the x to use for the operation

Examples

If you wanted to subtract 10 from all variables in a dataset, you would do the following:

```
>>> ds.subtract(10)
```

Or, you could use standard python subtraction syntax:

```
>>> ds - 10
```

To substract the values in a dataset ds2 from those in a dataset ds1, you would do the following:

```
>>> ds1.subtract(ds2)
```

Or, you could use standard python subtraction syntax:

```
>>> ds1 - ds2
```

Grids in the datasets must match. Division will occur in matching timesteps in ds1 and ds2 if there are matching timesteps. If there is only 1 timestep in ds2, then the data from that timestep in ds2 will be subtracted from the data in all timesteps in ds1.

nctoolkit.DataSet.divide

DataSet.**divide**(*self*, *x=None*, *var=None*)

divide: Divide the data.

This will divide the dataset by a constant, another dataset or a netCDF file.

Parameters

- **x** (*int, float, DataSet or netCDF file*) – An int, float, single file dataset or netCDF file to divide the dataset by. If a dataset or netCDF file is supplied, this must have only one variable, unless var is provided. The grids must be the same.
- **var** (*str*) – A variable in the x to use for the operation

Examples

If you wanted to divide all variables in a dataset by 20, you would do the following:

```
>>> ds.divide(10)
```

Or, you could use standard python division syntax:

```
>>> ds / 10
```

To divide values in a dataset by those in the dataset ds2 from a dataset ds1, you would do the following:

```
>>> ds1.divide(ds2)
```

Or, you could use standard python division syntax:

```
>>> ds1 / ds2
```

Grids in the datasets must match. Division will occur in matching timesteps in ds1 and ds2. If there is only 1 timestep in ds2, then the data from that timestep in ds2 will divided the data in all ds1 time steps.

4.23.15 Ensemble statistics

DataSet.ensemble_mean(self[, nco, ignore_time])	ensemble_mean: Calculate an ensemble mean
DataSet.ensemble_min(self[, nco, ignore_time])	ensemble_min: Calculate an ensemble minimum.
DataSet.ensemble_max(self[, nco, ignore_time])	ensemble_max: Calculate an ensemble maximum
DataSet.ensemble_percentile(self[, p])	ensemble_percentile: Calculate an ensemble percentile.
DataSet.ensemble_range(self)	ensemble_range: Calculate an ensemble range
DataSet.ensemble_stdev(self)	ensemble_stdev: Calculate an ensemble standard deviation
DataSet.ensemble_sum(self)	ensemble_sum: Calculate an ensemble sum
DataSet.ensemble_var(self)	ensemble_var: Calculate an ensemble variance

nctoolkit.DataSet.ensemble_mean

DataSet.**ensemble_mean**(*self*, *nco=False*, *ignore_time=False*)

ensemble_mean: Calculate an ensemble mean

This operates on a grid cell by grid cell basis.

Parameters

- **nco** (*boolean*) – Do you want to use NCO for the calculation? Default is False, i.e. CDO is used. Modify default if run time is an issue.
- **ignore_time** (*boolean*) – If True the mean is calculated over all time steps. If False, the ensemble mean is calculated for each time steps; for example, if the ensemble is made up of monthly files the mean for each month will be calculated.

Examples

If you had an ensemble of climate models with data covering the same time steps, you would calculate the ensemble mean as follows: ds.ensemble_mean()

If you had an ensemble of files that covered different time steps and want to calculate the mean over all time steps, you would do the following:

ds.ensemble_mean(ignore_time=True)

nctoolkit.DataSet.ensemble_min

DataSet.**ensemble_min**(*self*, *nco=False*, *ignore_time=False*)

ensemble_min: Calculate an ensemble minimum.

This operates on a grid cell by grid cell basis.

Parameters

- **nco** (*boolean*) – Do you want to use NCO for the calculation? Default is False, i.e. CDO is used. Modify default if run time is an issue.
- **ignore_time** (*boolean*) – If True the min is calculated over all time steps. If False, the ensemble min is calculated for each time steps; for example, if the ensemble is made up of monthly files the min for each month will be calculated.

Examples

If you had an ensemble of climate models with data covering the same time steps, you would calculate the ensemble min as follows:

```
>>> ds.ensemble_min()
```

nctoolkit.DataSet.ensemble_max

DataSet.**ensemble_max**(*self*, *nco=False*, *ignore_time=False*)

ensemble_max: Calculate an ensemble maximum

This operates on a grid cell by grid cell basis.

Parameters

- **nco** (*boolean*) – Do you want to use NCO for the calculation? Default is False, i.e. CDO is used. Modify default if run time is an issue.
- **ignore_time** (*boolean*) – If True the max is calculated over all time steps. If False, the ensemble max is calculated for each time steps; for example, if the ensemble is made up of monthly files the max for each month will be calculated.

Examples

If you had an ensemble of climate models with data covering the same time steps, you would calculate the ensemble max as follows:

```
>>> ds.ensemble_max()
```

nctoolkit.DataSet.ensemble_percentile

DataSet.**ensemble_percentile**(*self*, *p=None*)

ensemble_percentile: Calculate an ensemble percentile.

This will calculate the percentiles for each time step in the files. For example, if you had an ensemble of files where each file included 12 months of data, it would calculate the percentile for each month. This operates on a grid cell by grid cell basis.

Parameters **p** (*float or int*) – percentile to calculate. 0<=p<=100.

Examples

If you had an ensemble of climate models with data covering the same time steps, you would calculate the ensemble 90th percentile as follows:

```
>>> ds.ensemble_percentile(p=90)
```

nctoolkit.DataSet.ensemble_range

DataSet.**ensemble_range**(*self*)

ensemble_range: Calculate an ensemble range

The range is calculated for each time step; for example, if each file in the ensemble has 12 months of data the statistic will be calculated for each month.

This operates on a grid cell by grid cell basis.

Examples

If you had an ensemble of climate models with data covering the same time steps, you would calculate the ensemble range as follows:

```
>>> ds.ensemble_range()
```

nctoolkit.DataSet.ensemble_stdev

DataSet.**ensemble_stdev**(*self*)

ensemble_stdev: Calculate an ensemble standard deviation

The ensemble standard deviation is calculated for each time steps; for example, if the ensemble is made up of monthly files the standard deviation for each month will be calculated. This operates on a grid cell by grid cell basis.

Examples

If you had an ensemble of climate models with data covering the same time steps, you would calculate the ensemble standard deviation as follows:

```
>>> ds.ensemble_stdev()
```

nctoolkit.DataSet.ensemble_sum

DataSet.**ensemble_sum**(*self*)

ensemble_sum: Calculate an ensemble sum

The sum is calculated for each time step; for example, if each file in the ensemble has 12 months of data the statistic will be calculated for each month.

This operates on a grid cell by grid cell basis.

Examples

If you had an ensemble of climate models with data covering the same time steps, you would calculate the ensemble sum as follows:

```
>>> ds.ensemble_sum()
```

nctoolkit.DataSet.ensemble_var

DataSet.**ensemble_var**(*self*)

ensemble_var: Calculate an ensemble variance

The ensemble variance is calculated for each time steps; for example, if the ensemble is made up of monthly files the standard deviation for each month will be calculated. This operates on a grid cell by grid cell basis.

Examples

If you had an ensemble of climate models with data covering the same time steps, you would calculate the ensemble variance as follows:

```
>>> ds.ensemble_var()
```

4.23.16 Subsetting operations

DataSet.subset(self, **kwargs)	subset: A method for subsetting datasets to specific variables, years, longitudes etc.
DataSet.crop(self[, lon, lat, nco, nco_vars])	crop: Crop to a rectangular longitude and latitude box
DataSet.drop(self, **kwargs)	drop: Remove variables, days, months, years or time steps from a dataset

nctoolkit.DataSet.subset

DataSet.**subset**(*self*, ***kwargs*)

subset: A method for subsetting datasets to specific variables, years, longitudes etc.

Operations are applied in the order supplied.

Parameters ***kwargs** – Possible arguments: variables, years, months, seasons, timesteps, lon, lat

Note: this uses partial matches. So year, month, var etc. will also work

Each kwarg works as follows:

variables [str or list] A variable or list of variables to select. This method will accept wild cards. So using 'var*' would select all variables beginning with 'var'.

seasons [str] Seasons to select. One of "DJF", "MAM", "JJA", "SON".

days [list, range or int] Days(s) to select.

months [list, range or int] Month(s) to select.

years [list,range or int] Years(s) to select. These should be integers

hours [list, range or int] Hours(s) to select.

range [list] List of the form [date_min, date_max], where dates must be datetime objects or strings of the form "DD/MM/YYYY" or "DD-MM-YYYY". Times selected will be on or after date_min and before date_max.

timesteps [list or int] time step(s) to select. For example, if you wanted the first time step set times=0.

lon: list The longitude range to select. This must be two variables, between -180 and 180. The variables should be the minimum and maximum longitude.

lat: list The latitude range to select. This must be two variables, between -90 and 90. The variables should be the minimum and maximum latitude.

levels [list] List of the form [min_level, max_level]. Levels/depths between the two will be selected

Examples

If you want to select a single variable do the following:

```
>>> ds.subset(variable = "var")
```

If you want to select a list of variables, do this:

```
>>> ds.subset(variable = ["var1", "var2"])
```

If you want to select data for January, do the following:

```
>>> ds.subset(month = 1)
```

If you want to select a range of months, do the following:

```
>>> ds.subset(months = range(1, 7))
```

If you want to select a range of years, for example the 2010s, do the following:

```
>>> ds.subset(years = range(2010, 2020))
```

If you want to select the first two timesteps in a dataset, do the following:

```
>>> ds.subset(timesteps = [0,1])
```

nctoolkit.DataSet.crop

DataSet.**crop**(*self*, *lon=[- 180, 180]*, *lat=[- 90, 90]*, *nco=False*, *nco_vars=None*)

crop: Crop to a rectangular longitude and latitude box

Parameters

- **lon** (*list*) – The longitude range to select. This must be two variables, between -180 and 180 when nco = False.
- **lat** (*list*) – The latitude range to select. This must be two variables, between -90 and 90 when nco = False.
- **nco** (*boolean*) – Do you want this to use NCO for cropping? Defaults to False, and uses CDO. Set to True if you want to call NCO. NCO is typically better at handling very large horizontal grids.
- **nco_vars** (*str or list*) – If using NCO, the variables you want to select

Examples

If you wanted to crop a dataset to longitudes between -40 and 30 and latitudes between -10 and 40, you would do the following:

```
>>> ds.crop(lon = [-40, 30], lat = [-10, 40])
```

If you wanted to select only the northern hemisphere, the following will work:

```
>>> ds.crop(lat = [0, 90])
```

nctoolkit.DataSet.drop

DataSet.**drop**(*self*, ***kwargs*)

drop: Remove variables, days, months, years or time steps from a dataset

This will remove stated variables from files in the dataset.

Parameters ***kwargs** – Possible arguments: var, year, month, day

Note: this uses partial matches. So years, month, variable etc. will also work

Each kwarg works as follows:

var: str or list A variable or list of variables to select. This method will accept wild cards. So using 'var*' would select all variables beginning with 'var'.

day [list, range or int] Day(s) to drop.

month [list, range or int] Month(s) to drop.

year [list, range or int] Year(s) to drop.

time [list, range or int] Time steps to to drop. This can include negative indices.

Examples

If you wanted to remove a single variable 'var1' from a dataset data, you would do the following:

```
>>> ds.drop(variable = 'var')
```

If you wanted to remove a list of variables, you would do the following:

```
>>> ds.drop(variable = ['var1', 'var2', 'var2'])
```

If you wanted to remove the 29th Feburary you would do the following:

```
>>> ds.drop(month = 2, day = 29)
```

4.23.17 Time-based methods

DataSet.set_date(self[, year, month, day, ...])	Set the date in a dataset
DataSet.set_day(self, x)	Set the day for each time step in a dataset
DataSet.shift(self, **kwargs)	shift: Shift times in dataset by a number of hours, days, months, or years.

nctoolkit.DataSet.set_date

DataSet.**set_date**(*self*, *year=None*, *month=None*, *day=None*, *base_year=1900*)

Set the date in a dataset

You should only do this if you have to fix/change a dataset with a single, not multiple dates.

Parameters

- **year** (*int*) – The year
- **month** (*int*) – The month
- **day** (*int*) – The day
- **base_year** (*int*) – The base year for time creation in the netCDF. Defaults to 1900.

Examples

Set the date to 2000-01-01: >>> ds.set_date(year=2000, month=1, day=1)

nctoolkit.DataSet.set_day

DataSet.**set_day**(*self*, *x*)

Set the day for each time step in a dataset

Parameters **x** (*int*) – Day to set dataset to

Examples

Set the day to 1 for all time steps, effectively setting the date to the first of the month: >>> ds.set_day(1)

nctoolkit.DataSet.shift

DataSet.**shift**(*self*, ***kwargs*)

shift: Shift times in dataset by a number of hours, days, months, or years.

Operations are applied in the order supplied.

Parameters ***kwargs** – hours maps to shift_hours days maps to shift_days months maps to shift_months years maps to shift_years

Note: this uses partial matches. So hour, day, month, year will also work.

Examples

If you wanted to shift all times back 1 hour, you would do the following:

```
>>> ds.shift(hours = -1)
```

If you wanted to shift all times forward 2 days, you would do the following:

```
>>> ds.shift(days = 2)
```

If you wanted to shift all times forward 6 months, you would do the following:

```
>>> ds.shift(months = 6)
```

If you wanted to shift all times forward 1 year, you would do the following:

```
>>> ds.shift(years = 1)
```

This method will allow partial matches in arguments. So the following will do the same thing:

```
>>> ds.shift(year = 2)
```

```
>>> ds.shift(years = 2)
```

4.23.18 Interpolation, matching and resampling methods

DataSet.regrid(self[, grid, method, ...])	regrid: Regrid a dataset to a target grid
DataSet.to_latlon(self[, lon, lat, res, ...])	to_latlon: Regrid a dataset to a regular latlon grid
DataSet.match_points(self[, df, variables, ...])	match_points: Match dataset to a spatiotemporal points dataframe
DataSet.resample_grid(self[, factor])	resample_grid: Resample the horizontal grid of a dataset
DataSet.time_interp(self[, start, end, ...])	time_interp: Temporally interpolate variables based on date range and time resolution

continues on next page

Table 18 – continued from previous page

DataSet.timestep_interp(self[, steps])	timestep_interp: Temporally interpolate a dataset to given number of time steps between existing time steps
DataSet.fill_na(self[, n])	fill_na: Fill missing values with a distance-weighted average.
DataSet.box_mean(self[, x, y])	box_mean: Calculate the grid box mean for all variables.
DataSet.box_max(self[, x, y])	box_max: Calculate the grid box max for all variables.
DataSet.box_min(self[, x, y])	box_min: Calculate the grid box min for all variables.
DataSet.box_sum(self[, x, y])	box_sum: Calculate the grid box sum for all variables.
DataSet.box_range(self[, x, y])	box_range: Calculate the grid box range for all variables.

nctoolkit.DataSet.regrid

DataSet.**regrid**(*self*, *grid=None*, *method='bil'*, *recycle=False*, *one_grid=False*, ***kwargs*)

regrid: Regrid a dataset to a target grid

Horizontal interpolation

Parameters

- **grid** (*nctoolkit.DataSet, pandas data frame or netCDF file*) – The grid to remap to
- **method** (*str*) – Remapping method. Defaults to "bil". Methods available are: bilinear - "bil"; nearest neighbour - "nn" - "nearest neighbour" bicubic interpolation - "bic" Distance-weighted average - "dis" First order conservative remapping - "con" Second order conservative remapping - "con2" Large area fraction remapping - "laf"
- **recycle** (*bool*) – Set to True if you want to re-use the remapping weights when you are regridding another dataset.
- **one_grid** (*bool*) – Set to True if all files in multi-file dataset have the same grid, to speed things up.
- **kwargs** (*optional method to generate grid*) – Instead of supplying a grid using 'grid', you can supply *lon* and *lat*. These must be equally lengthed lists or arrays that will be used to generate the grid. If you want to regrid to a single location you can just supply a float to lon and lat.

Examples

Regrid to a grid defined by a pandas data frame:

```
>>> ds.regrid(grid=grid_df)
```

Regrid to a grid defined by a netCDF file:

```
>>> ds.regrid(grid="grid.nc")
```

Regrid to a grid defined by a nctoolkit.DataSet:

```
>>> ds.regrid(grid=grid_ds)
```

Regrid to a grid defined by a pandas data frame using nearest neighbour:

```
>>> ds.regrid(grid=grid_df, method="nn")
```

nctoolkit.DataSet.to_latlon

DataSet.**to_latlon**(*self*, *lon=None*, *lat=None*, *res=None*, *method='bil'*, *recycle=False*, *one_grid=False*)

to_latlon: Regrid a dataset to a regular latlon grid

Parameters

- **lon** (*list*) – 2 element list giving minimum and maximum longitude of target grid
- **lat** (*list*) – 2 element list giving minimum and maximum latitude of target grid
- **res** (*float, int or list*) – If float or int given, this will be the horizontal and vertical resolution of the target grid. If 2 element list is given, the first element is the longitudinal resolution and the second is the latitudinal resolution.
- **method** (*str*) – Remapping method. Defaults to "bil". Methods available are: bilinear - "bil"; nearest neighbour - "nn" - "nearest neighbour" bicubic interpolation - "bic" Distance-weighted average - "dis" First order conservative remapping - "con" Second order conservative remapping - "con2" Large area fraction remapping - "laf"
- **recycle** (*bool*) – Do you want the grid and weights to be available for recycling and use in regrid? Defaults to False
- **one_grid** (*bool*) – Set to True if all files in multi-file dataset have the same grid, to speed things up.

Examples

Regrid a dataset to a 0.25 degree latlon grid over the UK: >>> ds.to_latlon(lon=[-10, 10], lat=[50, 60], res=0.25)

Regrid a dataset to a 0.25 degree latlon grid over the UK, using nearest neighbour interpolation:

```
>>> ds.to_latlon(lon=[-10, 10], lat=[50, 60], res=0.25, method="nn")
```

nctoolkit.DataSet.match_points

DataSet.**match_points**(*self*, *df=None*, *variables=None*, *depths=None*, *tmean=False*, *top=False*, *nan=None*, *regrid='bil'*, *max_extrap=5*, ***kwargs*)

match_points: Match dataset to a spatiotemporal points dataframe

Parameters

- **df** (*pandas DataFrame*) – The column names must be made up of a subset of "lon", "lat", "year", "month", "day" and "depth" Pressure (in dbars), named "pressure", can also be used instead of "depth", which will require the optional dependency seawater to be installed.
- **variables** (*str or list*) – Str or list of variables. All variables are matched up if this is not supplied. This can include variables generated by assign using kwargs.
- **depths** (*nctoolkit DataSet or list*) – If each cell has different vertical levels, this must be provided as a dataset. If each cell has the same vertical levels, provide it as a list. If this is not supplied nctoolkit will try to figure out what they are. Only required if carrying out vertical matchups.

- **tmean** (*bool*) – Set to True or False, depending on whether you want temporal averaging at the temporal resolution given by df. For example, if you only had months in df, but had daily data in ds, you might want to calculate a daily average in the monthly dataset. This is equivalent to apply *ds.tmean(..)* to the dataset.
- **top** (*bool*) – Set to True if you want only the top/surface level of the dataset to be selected for matching.
- **nan** (*float or list*) – Value or range of values to set to nan. Defaults to 0. Only required if values in dataset need changed to missing
- **regrid** (*str*) – Regridding method. Defaults to "bil". Options available are those in nctoolkit regrid method. "nn" for nearest neighbour.
- **max_extrap** (*float*) – Maximum distance for extrapolation. Defaults to 5.
- **kwargs** (*kwargs*) – Additional arguments to send to assign

Returns matchpoints

Return type pandas.DataFrame

nctoolkit.DataSet.resample_grid

DataSet.**resample_grid**(*self*, *factor=None*)

resample_grid: Resample the horizontal grid of a dataset

Parameters **factor** (*int*) – The resampling factor. Must be a positive integer. No interpolation occurs. Example: factor of 2 will sample every other grid cell

Examples

If you wanted to select every other grid cell, you could do the following:

```
>>> ds.resample_grid(2)
```

nctoolkit.DataSet.time_interp

DataSet.**time_interp**(*self*, *start=None*, *end=None*, *resolution='monthly'*)

time_interp: Temporally interpolate variables based on date range and time resolution

Parameters

- **start** (*str*) – Start date for interpolation. Needs to be of the form YYYY/MM/DD or YYYY-MM-DD.
- **end** (*str*) – End date for interpolation. Needs to be of the form YYYY/MM/DD or YYYY-MM-DD. If end is not given interpolation will be to the final available time in the dataset.
- **resolution** (*str*) – Time steps used for interpolation. Needs to be "daily", "weekly", "monthly" or "yearly". Defaults to monthly.

Examples

Interpolate from 01/01/2000 to 01/01/2001 to monthly data:

```
>>> ds.time_interp(start="2000/01/01", end="2001/01/01", resolution="monthly")
```

Interpolate from 01/01/2000 to 01/01/2001 to daily data:

```
>>> ds.time_interp(start="2000/01/01", end="2001/01/01", resolution="daily")
```

Interpolate from 01/01/2000 to 01/01/2001 to weekly data:

```
>>> ds.time_interp(start="2000/01/01", end="2001/01/01", resolution="weekly")
```

nctoolkit.DataSet.timestep_interp

DataSet.**timestep_interp**(*self*, *steps=None*)

timestep_interp: Temporally interpolate a dataset to given number of time steps between existing time steps

Parameters **steps** (*int*) – Number of time steps to interpolate between existing time steps. For example, if you wanted to go from daily to hourly data you would set steps=24.

Examples

Interpolate from daily to hourly data:

```
>>> ds.timestep_interp(steps=24)
```

nctoolkit.DataSet.fill_na

DataSet.**fill_na**(*self*, *n=1*)

fill_na: Fill missing values with a distance-weighted average. This carries out infilling for each time step and vertical level.

Filling only uses horizontal neighbours, not vertical.

Parameters **n** (*int*) – Number of nearest neighbours to use. Defaults to 1. To

Examples

Fill missing values with a distance-weighted average using 5 nearest neighbours:

```
>>> ds.fill_na(n=5)
```

nctoolkit.DataSet.box_mean

DataSet.**box_mean**(*self*, *x=1*, *y=1*)
box_mean: Calculate the grid box mean for all variables.

This is performed for each time step.

Parameters

- **x** (*int*) – Number of boxes in the x, e.g. east-west direction
- **y** (*int or float*) – Number of boxes in the y, e.g. north-south direction

Examples

If you want to calculate the mean in each 4 by 4 box:

```
>>> ds.box_mean(x=4, y=4)
```

nctoolkit.DataSet.box_max

DataSet.**box_max**(*self*, *x=1*, *y=1*)
box_max: Calculate the grid box max for all variables.

This is performed for each time step.

Parameters

- **x** (*int*) – Number of boxes in the x, e.g. east-west direction
- **y** (*int or float*) – Number of boxes in the y, e.g. north-south direction

Examples

If you want to calculate the max in each 4 by 4 box:

```
>>> ds.box_max(x=4, y=4)
```

nctoolkit.DataSet.box_min

DataSet.**box_min**(*self*, *x=1*, *y=1*)
box_min: Calculate the grid box min for all variables.

This is performed for each time step.

Parameters

- **x** (*int*) – Number of boxes in the x, e.g. east-west direction
- **y** (*int or float*) – Number of boxes in the y, e.g. north-south direction

Examples

If you want to calculate the min in each 4 by 4 box:

```
>>> ds.box_min(x=4, y=4)
```

nctoolkit.DataSet.box_sum

DataSet.**box_sum**(*self*, *x=1*, *y=1*)

box_sum: Calculate the grid box sum for all variables.

This is performed for each time step.

Parameters

- **x** (*int*) – Number of boxes in the x, e.g. east-west direction
- **y** (*int or float*) – Number of boxes in the y, e.g. north-south direction

Examples

If you want to calculate the sum in each 4 by 4 box:

```
>>> ds.box_sum(x=4, y=4)
```

nctoolkit.DataSet.box_range

DataSet.**box_range**(*self*, *x=1*, *y=1*)

box_range: Calculate the grid box range for all variables.

This is performed for each time step.

Parameters

- **x** (*int*) – Number of boxes in the x, e.g. east-west direction
- **y** (*int or float*) – Number of boxes in the y, e.g. north-south direction

Examples

If you want to calculate the range in each 4 by 4 box:

```
>>> ds.box_range(x=4, y=4)
```

4.23.19 Masking methods

DataSet.mask_box(self[, lon, lat])	mask_box: Mask a lon/lat box

nctoolkit.DataSet.mask_box

DataSet.**mask_box**(*self*, *lon=[- 180, 180]*, *lat=[- 90, 90]*)
mask_box: Mask a lon/lat box

Parameters

- **lon** (*list*) – Longitude range to mask. Must be of the form: [lon_min, lon_max]
- **lat** (*list*) – Latitude range to mask. Must be of the form: [lat_min, lat_max]

Examples

If you want to mask a lon/lat box from -10 to 10 degrees longitude and -10 to 10 degrees latitude, you would do:
>>> data.mask_box(lon=[-10, 10], lat=[-10, 10])

4.23.20 Anomaly methods

DataSet.annual_anomaly(self[, baseline, ...])	annual_anomaly: Calculate annual anomalies for each variable based on a baseline period.
DataSet.monthly_anomaly(self[, baseline])	monthly:anomaly: Calculate monthly anomalies based on a baseline period.

nctoolkit.DataSet.annual_anomaly

DataSet.**annual_anomaly**(*self*, *baseline=None*, *metric='absolute'*, *window=1*, *align='right'*)
annual_anomaly: Calculate annual anomalies for each variable based on a baseline period.

The anomaly is derived by first calculating the climatological annual mean for the given baseline period. Annual means are then calculated for each year and the anomaly is calculated compared with the baseline mean. This will be calculated on a per-file basis in a multi-file dataset.

Parameters

- **baseline** (*list*) – Baseline years. This needs to be the first and last year of the climatological period. Example: a baseline of [1980,1999] will result in anomalies against the 20 year climatology from 1980 to 1999.
- **metric** (*str*) – Set to "absolute" or "relative", depending on whether you want the absolute or relative anomaly to be calculated.
- **window** (*int*) – A window for the anomaly. By default window = 1, i.e. the annual anomaly is calculated. If, for example, window = 20, the 20 year rolling means will be used to calculate the anomalies.

Examples

If you wanted to calculate an annual anomaly where values are compared with the mean for the years 1950-1969, you would do this:

```
>>> ds.annual_anomaly(baseline = [1950, 1969])
```

By default, this results in the absolute difference to be used. If you wanted the anomaly to be calculated relative to the baseline mean, you would do this:

```
>>> ds.annual_anomaly(baseline = [1950, 1969], metric = "relative")
```

You might want to smooth out the anomalies, so that you are looking at rolling averages. In that case you can supply a windows. So if you wanted to calculate the anomaly using a rolling average with a 10 year window, you would do this:

```
>>> ds.annual_anomaly(baseline = [1950, 1969], window = 10)
```

nctoolkit.DataSet.monthly_anomaly

DataSet.**monthly_anomaly**(*self*, *baseline=None*)

monthly:anomaly: Calculate monthly anomalies based on a baseline period.

The anomaly is derived by first calculating the climatological monthly mean for the given baseline period. Monthly means are then calculated for each year and the anomaly is calculated compared with the baseline mean. This is calculated separately for each file in a multi-file dataset.

Parameters **baseline** (*list*) – Baseline years. This needs to be the first and last year of the climatological period. Example: a baseline of [1985,2005] will result in anomolies against 20 year climatology from 1986 to 2005.

Examples

If you wanted to calculate a monthly anomaly where values are compared with the climatological monthly mean for the years 1950-1969, you would do this:

```
>>> ds.monthly_anomaly(baseline = [1950, 1969])
```

4.23.21 Statistical methods

DataSet.tmean(self[, over, align])	tmean: Calculate the temporal mean of all variables.
DataSet.tmin(self[, over, align])	tmin: Calculate the temporal minimum of all variables.
DataSet.tmedian(self[, over, align])	tmedian: Calculate the temporal median of all variables.
DataSet.tpercentile(self[, p, over, align])	tpercentile: Calculate the temporal percentile of all variables Useful for monthly percentile, annual/yearly percentile, seasonal percentile, daily percentile, daily climatology, monthly climatology, seasonal climatology
DataSet.tmax(self[, over, align])	tmax: Calculate the temporal maximum of all variables.
DataSet.tsum(self[, over, align])	tsum: Calculate the temporal sum of all variables.

continues on next page

Table 21 – continued from previous page

DataSet.trange(self[, over, align])	trange: Calculate the temporal range of all variables Useful for: monthly range, annual/yearly range, seasonal range, daily range, daily climatology, monthly climatology, seasonal climatology
DataSet.tstdev(self[, over, align])	tstdev: Calculate the temporal standard deviation of all variables Useful for: monthly standard deviation, annual/yearly standard deviation, seasonal standard deviation, daily standard deviation, daily climatology, monthly climatology, seasonal climatology
DataSet.tcumsum(self[, align])	tcumsum: Calculate the temporal cumulative sum of all variables
DataSet.tvar(self[, over, align])	tvar: Calculate the temporal variance of all variables Useful for: monthly variance, annual/yearly variance, seasonal variance, daily variance, daily climatology, monthly climatology, seasonal climatology
DataSet.cor_space(self[, var1, var2])	cor_space: Calculate the correlation correct between two variables in space.
DataSet.cor_time(self[, var1, var2])	cor_time: Calculate the correlation correct in time between two variables
DataSet.spatial_mean(self)	spatial_mean: Calculate the area weighted spatial mean for all variables.
DataSet.spatial_min(self)	spatial_min: Calculate the spatial minimum for all variables.
DataSet.spatial_max(self)	spatial_max: Calculate the spatial maximum for all variables.
DataSet.spatial_percentile(self[, p])	spatial_percentile: Calculate the spatial percentile for all variables
DataSet.spatial_range(self)	spatial_range: Calculate the spatial range for all variables.
DataSet.spatial_sum(self[, by_area])	spatial_sum: Calculate the spatial sum for all variables.
DataSet.spatial_stdev(self)	spatial_stdev: Calculate the spatial standard deviation for all variables.
DataSet.spatial_var(self)	spatial_var: Calculate the spatial variance for all variables.
DataSet.centre(self[, by, by_area])	centre: Calculate the latitudinal or longitudinal centre for each year/month combination in files.
DataSet.zonal_mean(self)	zonal_mean: Calculate the zonal mean for each time step
DataSet.zonal_min(self)	zonal_min: Calculate the zonal minimum for each time step
DataSet.zonal_max(self)	zonal_max: Calculate the zonal maximum for each time step
DataSet.zonal_range(self)	zonal_range: Calculate the zonal range for each time step
DataSet.zonal_sum(self[, by_area])	zonal_sum: Calculate the zonal sum for each time step
DataSet.meridonial_mean(self)	meridonial_mean: Calculate the meridonial mean for each year/month combination in files.
DataSet.meridonial_min(self)	meridonial_min: Calculate the meridonial minimum for each year/month combination in files.
DataSet.meridonial_max(self)	meridonial_max: Calculate the meridonial maximum for each year/month combination in files.

continues on next page

Table 21 – continued from previous page

DataSet.meridonial_range(self)	meridonial_range: Calculate the meridonial range for each year/month combination in files.

nctoolkit.DataSet.tmean

DataSet.**tmean**(*self*, *over='time'*, *align='right'*)

tmean: Calculate the temporal mean of all variables.

Useful for: monthly mean, annual/yearly mean, seasonal mean, daily mean, daily climatology, monthly climatology, seasonal climatology

Parameters

- **over** (*str or list*) – Time periods to average over. Options are 'year', 'month', 'day'. This operates in a similar way to the groupby method in pandas or the tidyverse in R.
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you want to calculate mean over all time steps. Do the following:

```
>>> ds.tmean()
```

If you want to calculate the mean for each year in a dataset, do this:

```
>>> ds.tmean("year")
```

If you want to calculate the mean for each month in each year in a dataset, do this:

```
>>> ds.tmean(["year", "month"])
```

This method will also let you easily calculate climatologies. So, if you wanted to calculate a monthly climatological mean, you would do this:

```
>>> ds.tmean( "month")
```

A daily climatological mean would be the following:

```
>>> ds.tmean( "day")
```

nctoolkit.DataSet.tmin

DataSet.**tmin**(*self*, *over='time'*, *align='right'*)

tmin: Calculate the temporal minimum of all variables.

Useful for: monthly minimum, annual/yearly minimum, seasonal minimum, daily minimum, daily climatology, monthly climatology, seasonal climatology

Parameters

- **over** (*str or list*) – Time periods to average over. Options are 'year', 'month', 'day'. This operates in a similar way to the groupby method in pandas or the tidyverse in R, with over acting as the grouping.

- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you want to calculate minimum over all time steps. Do the following:

```
>>> ds.tmin()
```

If you want to calculate the minimum for each year in a dataset, do this:

```
>>> ds.tmin("year")
```

If you want to calculate the minimum for each month in a dataset, do this:

```
>>> ds.tmin("month")
```

If you want to calculate the minimum for each month in each year in a dataset, do this:

```
>>> ds.tmin(["year", "month"])
```

This method will also let you easily calculate climatologies. So, if you wanted to calculate a monthly climatological min, you would do this:

```
>>> ds.tmin( "month")
```

A daily climatological minimum would be the following:

```
>>> ds.tmin( "day")
```

nctoolkit.DataSet.tmedian

DataSet.**tmedian**(*self*, *over='time'*, *align='right'*)

tmedian: Calculate the temporal median of all variables.

Useful for: monthly median, annual/yearly median, seasonal median, daily median, daily climatology, monthly climatology, seasonal climatology

Parameters

- **over** (*str or list*) – Time periods to average over. Options are 'year', 'month', 'day'. This operates in a similar way to the groupby method in pandas or the tidyverse in R, with over acting as the grouping.
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you want to calculate median over all time steps. Do the following:

```
>>> ds.tmedian()
```

If you want to calculate the median for each year in a dataset, do this:

```
>>> ds.tmedian("year")
```

If you want to calculate the median for each month in a dataset, do this:

```
>>> ds.tmedian("month")
```

If you want to calculate the median for each month in each year in a dataset, do this:

```
>>> ds.tmedian(["year", "month"])
```

This method will also let you easily calculate climatologies. So, if you wanted to calculate a monthly climatological median, you would do this:

```
>>> ds.tmedian( "month")
```

A daily climatological median would be the following:

```
>>> ds.tmedian( "day")
```

nctoolkit.DataSet.tpercentile

DataSet.**tpercentile**(*self*, *p=None*, *over='time'*, *align='right'*)

tpercentile: Calculate the temporal percentile of all variables Useful for monthly percentile, annual/yearly percentile, seasonal percentile, daily percentile, daily climatology, monthly climatology, seasonal climatology

Parameters

- **p** (*float or int*) – Percentile to calculate
- **over** (*str or list*) – Time periods to average over. Options are 'year', 'month', 'day'. This operates in a similar way to the groupby method in pandas or the tidyverse in R, with over acting as the grouping.
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you want to calculate the 20th percentile over all time steps. Do the following:

```
>>> ds.tpercentile(20)
```

If you want to calculate the 20th percentile for each year in a dataset, do this:

```
>>> ds.tpercentile(20)
```

If you want to calculate the 20th percentile for each year in a dataset, do this:

```
>>> ds.tpercentile(p= 20, over = "year")
```

nctoolkit.DataSet.tmax

DataSet.**tmax**(*self*, *over='time'*, *align='right'*)

tmax: Calculate the temporal maximum of all variables.

Useful for: monthly maximum, annual/yearly maximum, seasonal maximum, daily maximum, daily climatology, monthly climatology, seasonal climatology

Parameters

- **over** (*str or list*) – Time periods to average over. Options are 'year', 'month', 'day'. This operates in a similar way to the groupby method in pandas or the tidyverse in R, with over acting as the grouping.
- **= str** (*align*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you want to calculate maximum over all time steps. Do the following:

```
>>> ds.tmax()
```

If you want to calculate the maximum for each year in a dataset, do this:

```
>>> ds.tmax("year")
```

If you want to calculate the maximum for each month in a dataset, do this:

```
>>> ds.tmax("month")
```

If you want to calculate the maximum for each month in each year in a dataset, do this:

```
>>> ds.tmax(["year", "month"])
```

This method will also let you easily calculate climatologies. So, if you wanted to calculate a monthly climatological max, you would do this:

```
>>> ds.tmax( "month")
```

A daily climatological maximum would be the following:

```
>>> ds.tmax( "day")
```

nctoolkit.DataSet.tsum

DataSet.**tsum**(*self*, *over='time'*, *align='right'*)

tsum: Calculate the temporal sum of all variables.

Parameters

- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are “left”, “centre” and “right”
- **over** (*str or list*) – Time periods to count the sum over. Options are ‘year’, ‘month’, ‘day’. This operates in a similar way to the groupby method in pandas or the tidyverse in R, so you can supply combinations of these to get the sum over each year, month or day.

Examples

If you want to calculate sum over all time steps. Do the following: >>> ds.tsum() If you want to calculate the sum over each year: >>> ds.tsum(over=”year”) If you want to calculate the sum over each month. This will add up all data in each month across all years not within each year. >>> ds.tsum(over=”month”) If you want to calculate the sum over each day. This will add up all data in each day across all years not within each year. >>> ds.tsum(over=”day”)

nctoolkit.DataSet.trange

DataSet.**trange**(*self*, *over='time'*, *align='right'*)

trange: Calculate the temporal range of all variables Useful for: monthly range, annual/yearly range, seasonal range, daily range, daily climatology, monthly climatology, seasonal climatology

Parameters

- **over** (*str or list*) – Time periods to average over. Options are ‘year’, ‘month’, ‘day’. This operates in a similar way to the groupby method in pandas or the tidyverse in R, with over acting as the grouping.
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are “left”, “centre” and “right”

Examples

If you want to calculate range over all time steps. Do the following:

```
>>> ds.trange()
```

If you want to calculate the range for each year in a dataset, do this:

```
>>> ds.trange("year")
```

If you want to calculate the range for each month in a dataset, do this:

```
>>> ds.trange("month")
```

If you want to calculate the range for each month in each year in a dataset, do this:

```
>>> ds.trange(["year", "month"])
```

This method will also let you easily calculate climatologies. So, if you wanted to calculate a monthly climatological range, you would do this:

```
>>> ds.trange( "month")
```

A daily climatological range would be the following:

```
>>> ds.trange( "day")
```

nctoolkit.DataSet.tstdev

DataSet.**tstdev**(*self*, *over='time'*, *align='right'*)

tstdev: Calculate the temporal standard deviation of all variables Useful for: monthly standard deviation, annual/yearly standard deviation, seasonal standard deviation, daily standard deviation, daily climatology, monthly climatology, seasonal climatology

Parameters

- **over** (*str or list*) – Time periods to average over. Options are 'year', 'month', 'day'. This operates in a similar way to the groupby method in pandas or the tidyverse in R, with over acting as the grouping.
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you want to calculate standard deviation over all time steps. Do the following:

```
>>> ds.tstdev()
```

If you want to calculate the standard deviation for each year in a dataset, do this:

```
>>> ds.tstdev("year")
```

If you want to calculate the standard deviation for each month in a dataset, do this:

```
>>> ds.tstdev("month")
```

If you want to calculate the standard deviation for each month in each year in a dataset, do this:

```
>>> ds.tstdev(["year", "month"])
```

This method will also let you easily calculate climatologies. So, if you wanted to calculate a monthly climatological var, you would do this:

```
>>> ds.tstdev("month")
```

A daily climatological standard deviation would be the following:

```
>>> ds.tstdev("day")
```

nctoolkit.DataSet.tcumsum

DataSet.**tcumsum**(*self*, *align='right'*)

tcumsum: Calculate the temporal cumulative sum of all variables

Parameters **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you want to calculate the cumulative sum for all variables over all timesteps, do this:

```
>>> ds.tcumsum()
```

nctoolkit.DataSet.tvar

DataSet.**tvar**(*self*, *over='time'*, *align='right'*)

tvar: Calculate the temporal variance of all variables Useful for: monthly variance, annual/yearly variance, seasonal variance, daily variance, daily climatology, monthly climatology, seasonal climatology

Parameters

- **over** (*str or list*) – Time periods to average over. Options are 'year', 'month', 'day'. This operates in a similar way to the groupby method in pandas or the tidyverse in R, with over acting as the grouping.
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you want to calculate variance over all time steps. Do the following:

```
>>> ds.tvar()
```

If you want to calculate the variance for each year in a dataset, do this:

```
>>> ds.tvar("year")
```

If you want to calculate the variance for each month in a dataset, do this:

```
>>> ds.tvar("month")
```

If you want to calculate the variance for each month in each year in a dataset, do this:

```
>>> ds.tvar(["year", "month"])
```

This method will also let you easily calculate climatologies. So, if you wanted to calculate a monthly climatological var, you would do this:

```
>>> ds.tvar( "month")
```

A daily climatological variance would be the following:

```
>>> ds.tvar( "day")
```

nctoolkit.DataSet.cor_space

DataSet.**cor_space**(*self*, *var1=None*, *var2=None*)

cor_space: Calculate the correlation correct between two variables in space.

This is calculated for each time step. The correlation coefficient coefficient is calculated using values in all grid cells, ignoring missing values.

This calculates the Pearson correlation coefficient.

Parameters

- **var1** (*str*) – The first variable
- **var2** (*str*) – The second variable

Examples

If you wanted to calculate the spatial correlation coefficient between variables x and y in a dataset, you would do this:

```
>>> ds.cor_space("x", "y")
```

The correlation coefficient will be calculated for each time step.

nctoolkit.DataSet.cor_time

DataSet.**cor_time**(*self*, *var1=None*, *var2=None*)

cor_time: Calculate the correlation correct in time between two variables

The correlation is calculated for each grid cell, ignoring missing values. This calculates the Pearson correlation coefficient.

Parameters

- **var1** (*str*) – The first variable
- **var2** (*str*) – The second variable

Examples

If you wanted to calculate the temporal correlation coefficient between variables x and y in a dataset, you would do this:

```
>>> ds.cor_time("x", "y")
```

The correlation coefficient will be calculated for each grid cell. This method will indicate how temporally correlated variables are in different spatial regions.

nctoolkit.DataSet.spatial_mean

DataSet.**spatial_mean**(*self*)
spatial_mean: Calculate the area weighted spatial mean for all variables.

This is performed for each time step.

Examples

If you want to calculate the spatial mean for a dataset, just do the following:

```
>>> ds.spatial_mean()
```

Note: This method will calculate the average using weights based on each cell's area. If cell areas cannot be calculated, it will take a straight average, and a warning will say this.

---

nctoolkit.DataSet.spatial_min

DataSet.**spatial_min**(*self*)
spatial_min: Calculate the spatial minimum for all variables.

This is performed for each time step.

Examples

If you want to calculate the spatial minimum for a dataset, just do the following:

```
>>> ds.spatial_min()
```

nctoolkit.DataSet.spatial_max

DataSet.**spatial_max**(*self*)
spatial_max: Calculate the spatial maximum for all variables.

This is performed for each time step.

Examples

If you want to calculate the spatial maximum for a dataset, just do the following:

```
>>> ds.spatial_max()
```

nctoolkit.DataSet.spatial_percentile

DataSet.**spatial_percentile**(*self*, *p=None*)

spatial_percentile: Calculate the spatial percentile for all variables

This is performed for each time step.

Parameters **p** (*int or float*) – Percentile to calculate. 0<=p<=100.

Examples

If you want to calculate the median of each variable across space for a dataset, just do the following:

```
>>> ds.spatial_percentile(50)
```

nctoolkit.DataSet.spatial_range

DataSet.**spatial_range**(*self*)

spatial_range: Calculate the spatial range for all variables.

This is performed for each time step.

Examples

If you want to calculate the range of each variable across space for a dataset, just do the following:

```
>>> ds.spatial_max()
```

nctoolkit.DataSet.spatial_sum

DataSet.**spatial_sum**(*self*, *by_area=False*)

spatial_sum: Calculate the spatial sum for all variables.

This is performed for each time step.

Parameters **by_area** (*boolean*) – Set to True if you want to multiply the values by the grid cell area before summing over space. Default is False.

Examples

If you want to calculate the spatial sum each variable across space for a dataset, just do the following:

```
>>> ds.spatial_sum()
```

By default, this method simply sums up each grid cell value. In some cases this is not suitable. For example, the values in each cell may concentrations or values per square metre etc. In this case multiplying each cell value by the cell area is more suitable. Do the following:

```
>>> ds.spatial_sum(by_area = True)
```

Each cell's value will be multiplied by the area of the cell (in square metres) prior to calculating the spatial sum.

nctoolkit.DataSet.spatial_stdev

DataSet.**spatial_stdev**(*self*)

spatial_stdev: Calculate the spatial standard deviation for all variables.

This is performed for each time step.

Examples

If you want to calculate the spatial standard deviation for a dataset, just do the following:

```
>>> ds.spatial_stdev()
```

nctoolkit.DataSet.spatial_var

DataSet.**spatial_var**(*self*)

spatial_var: Calculate the spatial variance for all variables.

This is performed for each time step.

Examples

If you want to calculate the spatial variance for a dataset, just do the following:

```
>>> ds.spatial_var()
```

nctoolkit.DataSet.centre

DataSet.**centre**(*self*, *by='latitude'*, *by_area=False*)

centre: Calculate the latitudinal or longitudinal centre for each year/month combination in files.

This applies to each file in an ensemble.

Parameters

- **by** (*str*) – Set to 'latitude' if you want the latitudinal centre calculated. 'longitude' for longitudinal.
- **by_area** (*bool*) – If the variable is a value/m2 type variable, set to True, otherwise set to False.

Examples

If you want to calculate the latitudinal centre of a variable, accounting for grid cell area, you can do the following: >>> ds.centre(by='latitude', by_area=True)

If you want to calculate the longitudinal centre of a variable, you can do the following: >>> ds.centre(by='longitude', by_area=True)

nctoolkit.DataSet.zonal_mean

DataSet.**zonal_mean**(*self*)
zonal_mean: Calculate the zonal mean for each time step

Examples

If you want to calculate the zonal mean for a dataset, do the following:

```
>>> ds.zonal_mean()
```

nctoolkit.DataSet.zonal_min

DataSet.**zonal_min**(*self*)
zonal_min: Calculate the zonal minimum for each time step

Examples

If you want to calculate the zonal minimum for a dataset, do the following:

```
>>> ds.zonal_min()
```

nctoolkit.DataSet.zonal_max

DataSet.**zonal_max**(*self*)
zonal_max: Calculate the zonal maximum for each time step

Examples

If you want to calculate the zonal maximum for a dataset, do the following:

```
>>> ds.zonal_max()
```

nctoolkit.DataSet.zonal_range

DataSet.**zonal_range**(*self*)
zonal_range: Calculate the zonal range for each time step

Examples

If you want to calculate the zonal range for a dataset, do the following:

```
>>> ds.zonal_range()
```

nctoolkit.DataSet.zonal_sum

DataSet.**zonal_sum**(*self*, *by_area=False*)
zonal_sum: Calculate the zonal sum for each time step

Parameters **by_area** (*bool*) – Set to True if you want the cell value to be multiplied by the cell area prior to summing

Examples

If you want to calculate the zonal sum for a dataset, do the following:

```
>>> ds.zonal_sum()
```

nctoolkit.DataSet.meridonial_mean

DataSet.**meridonial_mean**(*self*)
meridonial_mean: Calculate the meridonial mean for each year/month combination in files.

This applies to each file in an ensemble.

Examples

If you want to calculate the meridonial mean for a dataset, do the following:

```
>>> ds.meridonial_mean()
```

nctoolkit.DataSet.meridonial_min

DataSet.**meridonial_min**(*self*)
meridonial_min: Calculate the meridonial minimum for each year/month combination in files.

This applies to each file in an ensemble.

Examples

If you want to calculate the meridonial minimum for a dataset, do the following:

```
>>> ds.meridonial_min()
```

nctoolkit.DataSet.meridonial_max

DataSet.**meridonial_max**(*self*)
meridonial_max: Calculate the meridonial maximum for each year/month combination in files.

This applies to each file in an ensemble.

Examples

If you want to calculate the meridonial maximum for a dataset, do the following:

```
>>> ds.meridonial_max()
```

nctoolkit.DataSet.meridonial_range

DataSet.**meridonial_range**(*self*)

meridonial_range: Calculate the meridonial range for each year/month combination in files.

This applies to each file in an ensemble.

Examples

If you want to calculate the meridonial range for a dataset, do the following:

```
>>> ds.meridonial_max()
```

4.23.22 Merging methods

DataSet.merge(self[, join, match, check])	merge: Merge a multi-file ensemble into a single file

nctoolkit.DataSet.merge

DataSet.**merge**(*self*, *join='variables'*, *match=['year', 'month', 'day']*, *check=True*)

merge: Merge a multi-file ensemble into a single file

2 methods are available. 1) merging files with different variables, but the same time steps. 2) merging files with the same variables, with different times.

Parameters

- **join** (*str*) – This defines the type of merging to carry out. "variables": this will merge by variable, so that an ensemble with different variables, but the same number of time steps is merged to a single file. "time": this will merge files with the same variables, but different times to a single file, into a single file with ordered times. join defaults to "variables", and uses partial matches, so "var" will give variable based merging.
- **match** (*list, str*) – Optional argument when join = 'variables'. A list or str stating what must match in the netCDF files. Defaults to year/month/day. This list must be some combination of year/month/day. An error will be thrown if the elements of time in match do not match across all netCDF files. The only exception is if there is a single date file in the ensemble.
- **check** (*bool*) – By default nctoolkit out checks in case files do not have the same variables etc. Set check to False if you are confident merging will be problem free. If you are unsure if files have the same variables, set check to True to find out. Note: if you do not explicitly provide check and there are more than 30 files in a dataset, checks will be turned off.

Examples

If you wanted to merge files with the same variables, but different time steps, you would do: >>> ds.merge(join='time') If you wanted to merge files with different variables, but the same time steps, you would do: >>> ds.merge(join='variables')

If you wanted to merge files with different variables, but the same time steps, but only needed to ensure that the month in each time step matched, you would do:

```
>>> ds.merge(join='variables', match='month')
```

The above may be useful if you have a dataset with monthly data, but some files have the first of the month, and some have the 15th of the month.

4.23.23 Splitting methods

DataSet.split(self[, by])	split: Split the dataset

nctoolkit.DataSet.split

DataSet.**split**(*self*, *by=None*)

split: Split the dataset

Each file in the ensemble will be separated into new files based on the splitting argument.

Parameters **by** (*str*) – Available by arguments are 'year', 'month', 'yearmonth', 'season', 'day' 'name', "timestep". year will split files by year, month will split files by month, yearmonth will split files by year and month; season will split files by year, day will split files by day. Using "timestep" will split files by timestep. 'name' will split by variable name

Examples

If you want to split each file into a dataset into a separate files for each year, do the following:

```
>>> ds.split("year")
```

If you wanted to split by month, do the following:

```
>>> ds.split("month")
```

4.23.24 Output and formatting methods

DataSet.to_nc(self, out[, zip, overwrite])	to_nc: Save a dataset to a named file.
DataSet.to_xarray(self[, decode_times])	to_xarray: Open a dataset as an xarray object
DataSet.to_dataframe(self[, decode_times])	to_dataframe: Convert a dataset to a pandas data frame
DataSet.zip(self)	zip: Zip the dataset
DataSet.format(self[, ext])	format: Change the netCDF format of a dataset.

nctoolkit.DataSet.to_nc

DataSet.**to_nc**(*self*, *out*, *zip=True*, *overwrite=False*, ***kwargs*)

to_nc: Save a dataset to a named file.

This will only work with single file datasets.

Parameters

- **out** (*str*) – Output file name.
- **zip** (*boolean*) – True/False depending on whether you want to zip the file. Default is True.
- **overwrite** (*boolean*) – If out file exists, do you want to overwrite it? Default is False.
- ****kwargs** (*kwargs*) – Optional arguments to be sent to subset.

Examples

If you want to export a dataset to a netCDF file, do the following:

```
>>> ds.to_nc("out.nc")
```

By default this file will be zipped. If you do not want it zipped, do this:

```
>>> ds.to_nc("out.nc", zip = False)
```

By default this cannot overwrite files. If the output file exists, do the following:

```
>>> ds.to_nc("out.nc", overwrite = True)
```

If you only want to export a subset of the data, you can use optional arguments that will be sent to subset. For example, if you only wanted the year 2000, you would do this:

```
>>> ds.to_nc("out.nc", year = 2000)
```

nctoolkit.DataSet.to_xarray

DataSet.**to_xarray**(*self*, *decode_times=True*, ***kwargs*)

to_xarray: Open a dataset as an xarray object

Parameters

- **decode_times** (*boolean*) – Set to False if you do not want xarray to decode the times. Default is True. If xarray cannot decode times, CDO will be used.
- ****kwargs** (*kwargs*) – Optional arguments to be sent to subset.

Returns to_xarray

Return type xarray.Dataset

Examples

If you want to convert a dataset to an xarray dataset, do the following:

```
>>> ds.to_xarray()
```

This will return an xarray dataset.

If you do not want time to be decoded, do the following:

```
>>> ds.to_xarray(decode_times = False)
```

nctoolkit.DataSet.to_dataframe

DataSet.**to_dataframe**(*self*, *decode_times=True*, ***kwargs*)

to_dataframe: Convert a dataset to a pandas data frame

Parameters

- **decode_times** (*boolean*) – Set to False if you do not want xarray to decode the times prior to conversion to data frame. Default is True.
- ****kwargs** (*kwargs*) – Optional arguments to be sent to subset.

Returns **to_dataframe**

Return type pandas.DataFrame

Examples

If you want to convert a dataset to a pandas data frame, do the following:

```
>>> ds.to_dataframe()
```

nctoolkit.DataSet.zip

DataSet.**zip**(*self*)

zip: Zip the dataset

This will compress the files within the dataset.

This will occur lazily, so will only occur after everything has been evaluated.

Examples

If you want to zip the files in a dataset, do the following:

```
>>> ds.zip()
```

nctoolkit.DataSet.format

DataSet.**format**(*self*, *ext=None*)

format: Change the netCDF format of a dataset.

This will compress the files within the dataset. This works lazily.

Parameters **ext** (*str*) – New format. Must be one of "nc", "nc1", "nc2", "nc4" and "nc5" . netCDF = nc1 netCDF version 2 (64-bit offset) = nc2/nc netCDF4 (HDF5) = nc4 netCDF4-classi = nc4c netCDF version 5 (64-bit data) = nc5

Examples

Change the format of a dataset to netCDF4: >>> ds.format("nc4")

Change the format of a dataset to netCDF5: >>> ds.format("nc5")

4.23.25 Miscellaneous methods

DataSet.na_count(self[, over, align])	na_count: Calculate the number of missing values.
DataSet.na_frac(self[, over, align])	na_frac: Calculate the fraction of missing values in each grid cell across all time steps.
DataSet.distribute(self[, m, n])	distribute: Split the dataset into multiple evenly sized horizontal and vertical new files
DataSet.collect(self)	Collect a dataset that has been split using distribute
DataSet.cell_area(self[, join])	cell_area: Calculate the area of grid cells.
DataSet.first_above(self[, x])	first_above: Identify the time step when a value is first above a threshold.
DataSet.first_below(self[, x])	first_below: Identify the time step when a value is first below a threshold This will do the comparison with either a number, a Dataset or a netCDF file.
DataSet.last_above(self[, x])	last_above: Identify the final time step when a value is above a threshold This will do the comparison with either a number, a Dataset or a netCDF file.
DataSet.last_below(self[, x])	last_below: Identify the last time step when a value is below a threshold This will do the comparison with either a number, a Dataset or a netCDF file.
DataSet.cdo_command(self[, command, ...])	cdo_command: Apply a cdo command
DataSet.nco_command(self[, command, ensemble])	Apply an nco command
DataSet.compare(self[, expression])	Compare all variables to a constant
DataSet.gt(self, x)	Method to calculate if variable in dataset is greater than that in another file or dataset This currently only works with single file datasets
DataSet.lt(self, x)	Method to calculate if variable in dataset is less than that in another file or dataset This currently only works with single file datasets
DataSet.reduce_dims(self)	reduce_dims: Reduce dimensions of data
DataSet.reduce_grid(self[, mask])	reduce_grid: Reduce the dataset to non-zero locations in a mask
DataSet.set_precision(self, x)	Set the precision in a dataset
DataSet.check(self)	check: Check contents of files for common data problems.

continues on next page

Table 25 – continued from previous page

DataSet.is_corrupt(self)	is_corrupt: Check if files are corrupt
DataSet.fix_nemo_ersem_grid(self)	A quick hack to change the grid file in North West European shelf Nemo grids.
DataSet.set_gridtype(self, grid)	Set the grid type.
DataSet.surface_mask(self)	surface_mask: Create a mask identifying the shallowest cell without missing values.
DataSet.strip_variables(self[, vars])	strip_variables: Remove any variables, such as bnds etc., from variables.
DataSet.no_leaps(self)	Remove leap years.
DataSet.as_double(self, x)	Set a variable/dimension to double This is mostly useful for cases when time is stored as an int, but you need a double
DataSet.as_type(self, x)	Set a variable/dimension to double This is mostly useful for cases when time is stored as an int, but you need a double
DataSet.reset(self)	Simple method to fully reset a datset

nctoolkit.DataSet.na_count

DataSet.**na_count**(*self*, *over='time'*, *align='right'*)

na_count: Calculate the number of missing values.

Parameters

- **over** (*str or list*) – Time periods to to the count over over. Options are 'time', 'year', 'month', 'day'.
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you want to calculate the number of missing values over all time steps. Do the following: >>> ds.na_count()
If you want to calculate the number of missing values in each year: >>> ds.na_count(over="year")

nctoolkit.DataSet.na_frac

DataSet.**na_frac**(*self*, *over='time'*, *align='right'*)

na_frac: Calculate the fraction of missing values in each grid cell across all time steps.

Parameters

- **over** (*str or list*) – Time periods to to the count over over. Options are 'time', 'year', 'month', 'day'.
- **align** (*str*) – This determines whether the output time is at the left, centre or right hand side of the time window. Options are "left", "centre" and "right"

Examples

If you want to calculate the fraction of missing values over all time steps. Do the following: >>> ds.na_frac() If you want to calculate the fraction of missing values in each year: >>> ds.na_frac(over="year")

nctoolkit.DataSet.distribute

DataSet.**distribute**(*self*, *m=1*, *n=1*)

distribute: Split the dataset into multiple evenly sized horizontal and vertical new files

Parameters

- **m** (*int*) – Number of rows
- **n** (*int*) – Number of columns

Examples

If you want to split the dataset into 2 by 2 with 4 new files: >>> ds.distribute(m=2, n=2)

nctoolkit.DataSet.collect

DataSet.**collect**(*self*)

Collect a dataset that has been split using distribute

Examples

```
>>> ds.distribute(4,4)
>>> #... Carry out some operations
>>> ds.collect()
```

nctoolkit.DataSet.cell_area

DataSet.**cell_area**(*self*, *join=True*)

cell_area: Calculate the area of grid cells.

Area of grid cells is given in square meters.

Parameters **join** (*boolean*) – Set to False if you only want the cell areas to be in the output. join=True adds the areas as a variable to the dataset. Defaults to True.

Examples

If you wanted to add the cell_areas as a new variable in a dataset, you would do the following:

```
>>> ds.cell_area()
```

If you wanted to replace a dataset with the cell areas of that dataset, you would do the following:

```
>>> ds.cell_area(join = False)
```

nctoolkit.DataSet.first_above

DataSet.**first_above**(*self*, *x=None*)

first_above: Identify the time step when a value is first above a threshold.

This will do the comparison with either a number, a Dataset or a netCDF file.

Parameters **x** (*int, float, DataSet or netCDF file*) – An int, float, single file dataset or netCDF file to use for the threshold(s). If comparing with a dataset or single file there must only be a single variable in it. The grids must be the same.

Examples

If you wanted to calculate the first time step where the value in a grid cell goes above 10, you would do the following

```
>>> ds.first_above(10)
```

If you wanted to calculate the first time step where the value in a grid cell goes above that in another dataset, the following will work. Note that both datasets must have the same grid, and can only have single variables. The second dataset can, of course, only have one timestep.

```
>>> ds.first_above(ds1)
```

nctoolkit.DataSet.first_below

DataSet.**first_below**(*self*, *x=None*)

first_below: Identify the time step when a value is first below a threshold This will do the comparison with either a number, a Dataset or a netCDF file.

Parameters **x** (*int, float, DataSet or netCDF file*) – An int, float, single file dataset or netCDF file to use for the threshold(s). If comparing with a dataset or single file there must only be a single variable in it. The grids must be the same.

Examples

If you wanted to calculate the first time step where the value in a grid cell goes below 10, you would do the following

```
>>> ds.first_below(10)
```

If you wanted to calculate the first time step where the value in a grid cell goes above that in another dataset, the following will work. Note that both datasets must have the same grid, and can only have single variables. The second dataset can, of course, only have one timestep.

```
>>> ds.first_below(ds1)
```

nctoolkit.DataSet.last_above

DataSet.**last_above**(*self*, *x=None*)

last_above: Identify the final time step when a value is above a threshold This will do the comparison with either a number, a Dataset or a netCDF file.

Parameters **x** (*int, float, DataSet or netCDF file*) – An int, float, single file dataset or netCDF file to use for the threshold(s). If comparing with a dataset or single file there must only be a single variable in it. The grids must be the same.

Examples

If you wanted to calculate the last time step where the value in a grid cell is above 10, you would do the following

```
>>> ds.first_above(10)
```

If you wanted to calculate the last time step where the value in a grid cell goes above that in another dataset, the following will work. Note that both datasets must have the same grid, and can only have single variables. The second dataset can, of course, only have one timestep.

```
>>> ds.first_above(ds1)
```

nctoolkit.DataSet.last_below

DataSet.**last_below**(*self*, *x=None*)

last_below: Identify the last time step when a value is below a threshold This will do the comparison with either a number, a Dataset or a netCDF file.

Parameters **x** (*int, float, DataSet or netCDF file*) – An int, float, single file dataset or netCDF file to use for the threshold(s). If comparing with a dataset or single file there must only be a single variable in it. The grids must be the same.

Examples

If you wanted to calculate the last time step where the value in a grid cell is below 10, you would do the following

```
>>> ds.last_below(10)
```

If you wanted to calculate the last time step where the value in a grid cell is above that in another dataset, the following will work. Note that both datasets must have the same grid, and can only have single variables. The second dataset can, of course, only have one timestep.

```
>>> ds.last_below(ds1)
```

nctoolkit.DataSet.cdo_command

DataSet.**cdo_command**(*self*, *command=None*, *ensemble=False*, *check=False*)
cdo_command: Apply a cdo command

Parameters

- **command** (*string*) – cdo command to call. This command must be such that "cdo {command} infile outfile" will run.
- **ensemble** (*bool*) – Is this an ensemble method command? For example ensmean, mergetime, etc.
- **check** (*bool*) – Check whether the command is valid

Examples

Use CDO to select a year from a dataset >>> ds.cdo_command("selyear,2000")

nctoolkit.DataSet.nco_command

DataSet.**nco_command**(*self*, *command=None*, *ensemble=False*)
Apply an nco command

Parameters

- **command** (*string*) – nco command to call. This must be of a form such that "nco {command} infile outfile" will run.
- **ensemble** (*boolean*) – Set to True if you want the command to take all of the files as input. This is useful for ensemble methods.

Examples

Select a variable from a file >>> ds.nco_command("ncks -v tas")

nctoolkit.DataSet.compare

DataSet.**compare**(*self*, *expression=None*)
Compare all variables to a constant

Parameters **expression** (*str*) – This a regular comparison such as "<0", ">0", "==0"

Examples

If you wanted to identify grid cells with positive values you would do the following:

```
>>> ds.compare(">0")
```

This could also be done using standard python syntax:

```
>>> ds > 0
```

This will be calculcated for each time step.

If you wanted to identify grid cells with negative values, you would do this

```
>>> ds.compare("<0")
```

This could also be done using standard python syntax:

```
>>> ds < 0
```

nctoolkit.DataSet.gt

DataSet.**gt**(*self*, *x*)

Method to calculate if variable in dataset is greater than that in another file or dataset This currently only works with single file datasets

Parameters **x** (*str or single file dataset*) – File path or nctoolkit dataset

nctoolkit.DataSet.lt

DataSet.**lt**(*self*, *x*)

Method to calculate if variable in dataset is less than that in another file or dataset This currently only works with single file datasets

Parameters **x** (*str or single file dataset*) – File path or nctoolkit dataset

nctoolkit.DataSet.reduce_dims

DataSet.**reduce_dims**(*self*)

reduce_dims: Reduce dimensions of data

This will remove any dimensions with only one value. For example, if only selecting one vertical level, the vertical dimension will be removed.

Examples

If you want to remove any dimensions that have only one value, do the following:

```
>>> ds.reduce_dims("out.nc")
```

Note that this will work lazily. This method is most useful when you want to simplify datasets before exporting them to something like a pandas dataframe.

nctoolkit.DataSet.reduce_grid

DataSet.**reduce_grid**(*self*, *mask=None*)

reduce_grid: Reduce the dataset to non-zero locations in a mask

Parameters **mask** (*str or dataset*) – single variable dataset or path to .nc file. The mask must have an identical grid to the dataset.

nctoolkit.DataSet.set_precision

DataSet.**set_precision**(*self*, *x*)
Set the precision in a dataset

Parameters **x** (*str*) – The precision. One of 'I8', 'I16', 'I32', 'F32', 'F64'.

Examples

Set the precision to 32 bit: >>> ds.set_precision("F32")

nctoolkit.DataSet.check

DataSet.**check**(*self*)
check: Check contents of files for common data problems.

Example

Check if files have any issues: >>> ds.check()

nctoolkit.DataSet.is_corrupt

DataSet.**is_corrupt**(*self*)
is_corrupt: Check if files are corrupt

Example

```
>>> ds.is_corrupt()
```

nctoolkit.DataSet.fix_nemo_ersem_grid

DataSet.**fix_nemo_ersem_grid**(*self*)
A quick hack to change the grid file in North West European shelf Nemo grids.

nctoolkit.DataSet.set_gridtype

DataSet.**set_gridtype**(*self*, *grid*)
Set the grid type. Only use this if, for example, the grid is "generic" when it should be lonlat.

Parameters **grid** (*str*) – Grid type. Needs to be one of "curvilinear", "unstructured", "dereference", "regular", "regularnn" or "lonlat".

nctoolkit.DataSet.surface_mask

DataSet.**surface_mask**(*self*)

surface_mask: Create a mask identifying the shallowest cell without missing values.

This converts a dataset to a mask identifying which cell represents top level, for example the sea surface. 1 identifies the shallowest cell with non-missing values. Everything else is 0, or missing. At present this method only uses the first available variable from netCDF files, so it may not be suitable for all data

Examples

If you wanted to create a mask identifying the surface, you would do this:

```
>>> ds.surface_mask()
```

nctoolkit.DataSet.strip_variables

DataSet.**strip_variables**(*self*, *vars=None*)

strip_variables: Remove any variables, such as bnds etc., from variables.

This should probably only be done at the end of a processing chain before converting to a dataframe etc., as it is stripping away critical info for netCDF operations.

Parameters **vars** (*str or list*) – individual or list of variables to select and strip. All variables will be stripped if this is not defined.

nctoolkit.DataSet.no_leaps

DataSet.**no_leaps**(*self*)

Remove leap years. This uses an undocumented CDO feature to remove Feb 29 and sets the calendar to leap year free

Examples

Remove leap years from a dataset: >>> ds.no_leaps()

nctoolkit.DataSet.as_double

DataSet.**as_double**(*self*, *x*)

Set a variable/dimension to double This is mostly useful for cases when time is stored as an int, but you need a double

Parameters **x** (*list*) – A list of variable/dimensions you want to convert to floats

Examples

Change time to double: >>> ds.as_double('time])

nctoolkit.DataSet.as_type

DataSet.**as_type**(*self*, *x*)

Set a variable/dimension to double This is mostly useful for cases when time is stored as an int, but you need a double

Parameters **x** (*dict*) – A dictionary mapping variables to type. Values in dict must be one of 'int', 'float32' and 'float64'.

Examples

Change time to float64: >>> ds.as_type({'time': 'float64'})

nctoolkit.DataSet.reset

DataSet.**reset**(*self*)

Simple method to fully reset a datset

4.23.26 Ecological methods

DataSet.phenology(self[, var, metric, p])	phenology: Calculate phenologies from a dataset

nctoolkit.DataSet.phenology

DataSet.**phenology**(*self*, *var=None*, *metric=None*, *p=None*)

phenology: Calculate phenologies from a dataset

Each file in an ensemble must only cover a single year, and ideally have all days. The method assumes datasets have daily resolution.

Parameters

- **var** (*str*) – Variable to analyze.
- **metric** (*str*) – Must be peak, middle, start or end. Peak is defined as the day of the maximum value. Middle is the day when the cumulative total of the variable first exceeds the cumulative total for the entire year. Start or end is defined as the first day when the cumulative total exceeds a percentile p of the maximum cumulative total.
- **p** (*str*) – Percentile to use for start or end.

4.24 Contributing to nctoolkit

We welcome contributions to nctoolkit! The following are all welcome contributions:

- Bug reports
- Bug fixes
- New features
- New documentation
- New examples
- New tutorials
- New tests

Right now, nctoolkit is developed by marine scientists, and we would love to have more input from other fields. In particular, we would love to have more input from the atmospheric sciences community. If you are interested in contributing, please reach out to us!

4.24.1 Report bugs using Github's issues

We track bugs using Github's issues. If you have found a bug, please open an issue.

If you do raise an issue try to do the following:

- Check if the issue has already been raised
- Check if the issue has already been fixed in the latest code
- Create a reproducible example that demonstrates the problem
- Specify the operating system you are using and Python version
- Specify the version of nctoolkit you are using

If you are able to fix the bug yourself, please open a pull request with the fix.

The developers will try to respond to issues as quickly as possible, but please be patient.

4.24.2 All contributions you make will be under the GNU General Public License v3.0

When you submit code changes, your submissions are understood to be under the same GNU General Public License v3.0. Feel free to contact the maintainers if that's a concern.

As stated in the license, we are not liable for any damages that may arise from your use of the code. Read the full license here.

4.24.3 How to suggest a feature or enhancement or contribute code

Our preferred work flow for suggesting a new feature is to first either open an issue or a new discussion on Github. This allows us to discuss the feature before you spend time writing code.

Naturally, the best way to get your feature accepted is to write it yourself. When you are ready to write code, please follow the following steps:

1. Fork the repo and create your branch from *master*.
2. If you've added code that should be tested, add tests.
3. If you've changed APIs, update the documentation.
4. Ensure the test suite passes.
5. Make sure your code lints.
6. Issue that pull request!

We will try to respond to pull requests as quickly as possible, but please be patient.

4.24.4 A note on nctoolkit methods

In general, dataset methods do one of two things: modifying datasets or analyzing datasets. Analysis will include things such as plotting. Modifying datasets should always results in the temporary file(s) associated with the dataset being updated.

As explained in the documentation, methods should be as lazy as possible. This allows nctoolkit to chain CDO commands together and prevents unnecessary I/O. If you have a method that is not lazy, we can discuss if it can be made so.

Ideally, method code will either call CDO using the *cdo_command* method or use existing nctoolkit methods.

If you want to get started with the API a good place to start is the *fill_na* method.

```
def fill_na(self, n=1):
    """
    Fill missing values with a distance-weighted average. This carries out infilling for␣
↪each time step and vertical level.

    Parameters
    -------------
    n: int
        Number of nearest neighbours to use. Defaults to 1. To
    """

    cdo_command = f"cdo -setmisstodis,{n}"

    self.cdo_command(command=cdo_command, ensemble=False)
```

This method fills missing values using distance weighting. The CDO call is equivalent of the following:

```
$ cdo -setmisstodis,n input.nc output.nc
```

where n is the number of nearest neighbours to use. If the method you are considering writing can be implemented using CDO, then it is best to use CDO under the hood.

If you are unfamiliar with CDO, it is best to look through their excellent user guide.

At present, there are many methods in CDO that have yet to be implemented in nctoolkit. This includes EOFs and trend analysis. If you use CDO and nctoolkit and you feel something exists in CDO that should be in nctoolkit, reach out or open an issue.

Methods should be placed in an appropriate Python file in the *nctoolkit* directory. This should either be in an existing file if the method sits well alongside existing methods, or in a new file if it is a new category of methods.

Once you have added the method code, you will need to ensure it is imported. This can be done at the bottom of the *nctoolkit/api.py* file. For example, the *fill* method is imported as follows:

```
from nctoolkit.fill import fill_na
```

In addition to CDO, if you want to implement a method that uses NCO you can do so using the *nco_command* method. In effect, nctoolkit is capable of doing anything CDO or NCO can. So there are many opportunities to contribute.

4.25 Package info

This package was created by Robert Wilson at Plymouth Marine Laboratory (PML).

4.25.1 Acknowledgements

The current codebase of nctoolkit was developed using funding from the NERC Climate Linked Atlantic Sector Science programme (NE/R015953/1) and a combination of UK Research and Innovation (UKRI) and European Research Council (ERC) funded research projects.

4.25.2 Bugs and issues

If you identify bugs or issues with the package please raise an issue at PML's Marine Systems Modelling group's GitHub page here or contact nctoolkit's creator at rwi@pml.ac.uk.

4.25.3 Contributions welcome

The package is new, with new features being added each month. There remain a large number of features that could be added, especially for dealing with atmospheric data. If packages users are interested in contributing or suggesting new features they are welcome to raise and issue at the package's GitHub page or contact me.

4.26 Cheat sheet

A cheat sheet providing a quick 2-page overview of nctoolkit is available here.

INDEX

Y

Z