

---

# **nctoolkit**

**Robert Wilson**

**Dec 06, 2022**



# QUICK OVERVIEW

1	Who is nctoolkit for?	3
2	What type of data is nctoolkit designed for?	5
3	What systems can nctoolkit work on?	7
4	What can nctoolkit do?	9
	Index	119



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



## **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.





## **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.



## WHAT SYSTEMS CAN NCTOOLKIT WORK ON?

nctoolkit requires a Linux or macOS operating system.



## 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.

nctoolkit is available from the [Python Packaging Index](#). To install nctoolkit using pip:

```
$ pip install numpy
$ pip install nctoolkit
```

If you already have numpy installed, ignore the first line. This is only included as it will make installing some dependencies smoother. nctoolkit partly relies on cartopy for plotting. This has some additional dependencies, so you may need to follow their [guide](#) to ensure cartopy is installed fully. If you install nctoolkit using conda, you will not need to worry about that.

If you install nctoolkit from pypi, you will need to install the system dependencies listed below.

nctoolkit can also be installed using conda, as follows:

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

Note that recent releases are not available for Python 3.8 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 and Python 3.8, it is best to install using pip.

At present this can be slow due to the time taken to resolve dependency versions. If you run into problems just use pip.

To install the development version from GitHub:

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

### 4.1.2 Jupyter notebook issue

A recent update to ipykernel has broken some functionality in jupyter notebooks. This appears to be impacting plotting in nctoolkit. To fix this just downgrade ipykernel:

```
$ conda install ipykernel=6.17.1
$ mamba install ipykernel=6.17.1
$ pip install ipykernel==6.17.1
```

### 4.1.3 Python dependencies

- Python (3.6 or later)
- [numpy](#) (1.14 or later)
- [pandas](#) (0.24 or later)
- [xarray](#) (0.14 or later)
- [netCDF4](#) (1.53 or later)
- [ncplot](#)

### 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
```

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 to carry out almost all analysis and post-processing chains, and to do so easily and efficiently. 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:

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

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

### 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:

```
[2]: ds = nc.open_data("sst.mon.mean.nc")
      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    [time]
      :Overlay
      .Image.I      :Image    [lon,lat]    (sst)
      .Coastline.I  :Feature   [Longitude, Latitude]
```

## 4.2.2 It lets you calculate spatial averages

Calculating the spatial mean

```
[3]: ds = nc.open_data("sst.mon.mean.nc")
      ds.spatial_mean()
      ds.plot()
```

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

## 4.2.3 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.

```
[4]: 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()
```

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

## 4.2.4 It lets you crop data

We can crop to a specific region using the crop method. To get a region covering most of Europe, we could do this:

```
[5]: ds = nc.open_data("sst.mon.mean.nc")
      ds.crop(lon = [-13, 38], lat = [30, 67])
      ds.plot()
```

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

## 4.2.5 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 lets you regrid to a regular latlon grid. You just need to specify the extent of the new grid, the resolution and the regridding method.

```
[6]: ds = nc.open_data("sst.mon.mean.nc")
      ds.to_latlon(lon = [-13, 38], lat = [30, 67], res = 0.5, method = "nn")
      ds.plot()
```



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

## 4.2.6 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:

```
[7]: ds = nc.open_data("sst.mon.mean.nc")
      ds.trange("year")
      ds.tmean()
      ds.plot()

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

## 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.

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

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

## 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:

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

[9]: :QuadMesh      [time,lat]    (sst)
```

## 4.2.9 Getting started with nctoolkit

To get started with nctoolkit it is best to start [here](#), and to consider getting the [cheatsheet](#).

## 4.3 News

### 4.3.1 Release of v0.8.4

Version 0.8.4 will be released in December 2022.

This update will improve the ability to handle missing values. A method will be introduced for changing the fill value missing values, *set\_fill*.

Another method *missing\_as* is introduced. This will be 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.

### 4.3.2 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 installations 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.

### 4.3.3 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.

#### 4.3.4 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 dataset 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
```

#### 4.3.5 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')*.

#### 4.3.6 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.

### 4.3.7 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.

### 4.3.8 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.

### 4.3.9 Release of v0.5.0

This release was made on 13th June 2022. The *match\_points* method now allows extrapolation to vertical depths.

### 4.3.10 Release of v0.4.9

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

### 4.3.11 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.

### 4.3.12 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.

### 4.3.13 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.

### 4.3.14 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.

### 4.3.15 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 `tvvariance`` have been removed after periods of deprecation. Use `top`, `merge` and `tvar` instead.

### 4.3.16 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 longer 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")`.

### 4.3.17 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`.

### 4.3.18 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.

### 4.3.19 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, multiplying and subtracting 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 monthly 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.

### 4.3.20 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.

### 4.3.21 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 occasionally throw an error.

### 4.3.22 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.

### 4.3.23 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 `tvvariance` will be deprecated and is now renamed `tvar` for naming consistency.

### 4.3.24 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`.

### 4.3.25 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`.



### 4.3.26 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.

### 4.3.27 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.

### 4.3.28 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 Datasets

### 4.4.1 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](#).

### 4.4.2 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.1tm.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
```

### 4.4.3 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 compatibility. 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()
```

#### 4.4.4 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.1tm.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.

### 4.4.5 Lazy evaluation by default

Look at the processing chain below.

```
[ ]: ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.1tm.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.1tm.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.1tm.1981-
    ↪2010.nc")
ds.subset(time = 0)
ds.plot()
```

### 4.4.6 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)
```

### 4.4.7 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
```

## 4.5 Importing and exporting data

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

### 4.5.1 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")
```

### 4.5.2 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)
```

### 4.5.3 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)
```

## 4.5.4 Exporting datasets

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

### 4.5.5 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)
```

### 4.5.6 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()
```

### 4.5.7 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:

### 4.5.8 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])
```

## 4.6 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.

### 4.6.1 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")
```

### 4.6.2 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))
```

### 4.6.3 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])
```



### 4.6.4 Selecting seasons

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

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

### 4.6.5 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))
```

### 4.6.6 Geographic subsetting

If you want to select a geographic subregion of a dataset, you can use `crop`. 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.crop(lon = [-80, 90], lat = [50, 80])
```

Alternatively, you can use the `subset` method:

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

## 4.7 Interpolation

nctoolkit features built in methods for horizontal and vertical interpolation.

### 4.7.1 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 dataset [here](#).

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

```
[1]: 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()
```

nctoolkit is using Climate Data Operators version 1.9.10

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

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

## 4.7.2 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.

```
[2]: 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()
```

```
[2]:
```

		lon	lat	sst
time	ncells			
1850-01-01	0	-30.0	50.0	10.935501

## 4.7.3 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 2 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:

```
[3]: 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()
```

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

### 4.7.4 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 illustrate this with a dataset of global sea surface temperature. Let's start by cropping this dataset to the northern hemisphere.

```
[4]: 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()

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

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

```
[5]: 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()

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

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.

### 4.7.5 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`.

```
[6]: 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()

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

[7]: 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()
```

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

### 4.7.6 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 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.

```
[8]: 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()
```

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

### 4.7.7 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:

```
[9]: 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()
```

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

### 4.7.8 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.

```
[10]: ds = nc.open_thredds("https://data.nodc.noaa.gov/thredds/dodsC/ncei/woa/temperature/A5B7/
      ↪1.00/woa18_A5B7_t01_01.nc")
      ds.subset(variables="t_an")
      ds.vertical_interp(levels= [500])
      ds.plot()
```

```
Warning: xarray could not decode times!
```

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

```
[ ]:
```

## 4.8 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 illustrate 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:

```
[1]: import nctoolkit as nc
      ff = "sst.mon.mean.nc"
      ds = nc.open_data(ff)
      ds.subset(year = 2000)
      ds.tmean()
      ds.plot()
```

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

```
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
```

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

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

```
[2]: ff = "sst.mon.mean.nc"
      ds = nc.open_data(ff)
      ds.subset(year = 2000)
      ds.tmean()
```

(continues on next page)

(continued from previous page)

```
ds.zonal_mean()
ds.plot()
```

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

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

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

```
[3]: :QuadMesh    [time,lat]    (sst)
```

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

```
[4]: ff = "sst.mon.mean.nc"
ds = nc.open_data(ff)
ds.spatial_mean()
ds.plot()
```

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

### 4.8.1 Internal: ncplot

Plotting is carried out using the ncplot package. If you come across any errors, please raise an issue [here](#).

This is a package that aims to deliver easy use. Colour scales for heat map default to a diverging blue-to-red palette when there are positives and negatives and a viridis palette otherwise.

## 4.9 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"])
```

### 4.9.1 Calculating rolling averages

nctoolkit has a range of methods to calculate 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 calculate a rolling weekly sum, this would do:

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

### 4.9.2 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.

### 4.9.3 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")
```

### 4.9.4 Calculating climatologies

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

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

### 4.9.5 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.

## 4.10 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.



### 4.10.1 Merging multi-file datasets

There are two ways to merge multi-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.

### 4.10.2 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(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.

### 4.10.3 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`.

## 4.11 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.

### 4.11.1 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/nci/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.

### 4.11.2 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 then interpolated vertically with 1d interpolation using `scipy`.

### 4.11.3 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.

### 4.11.4 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.

## 4.12 Mathematical operations

### 4.12.1 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 unfamiliar 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)
```

## 4.12.2 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))
```

## 4.12.3 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)
```

#### 4.12.4 Functions that work with nctoolkit variables

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

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

### 4.12.5 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.

### 4.12.6 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")
```

## 4.13 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/nci/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 away 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
```

### 4.13.1 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]
```

### 4.13.2 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 specify 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]
```

### 4.13.3 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]
```

### 4.13.4 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.

## 4.14 Simple arithmetic and comparisons

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

### 4.14.1 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 were 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)
```

## 4.14.2 Comparisons

As with simple arithmetic, 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 oceans 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 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%.

## 4.15 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 package to process files in parallel using that chain.

### 4.15.1 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.

### 4.15.2 Parallel processing using multiprocessing

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 use multiprocessing to do the following: take all of the files in folder foo, do a bunch of things to them, then save the results in a new folder:

We start with a function giving a processing chain. There are obviously different ways of doing this, but I like to use a function that takes the input file and output file:

```
def process_chain(infile, outfile):
    ds = nc.open_data(ff)
    ds.assign(tos = lambda x: x.sst + 273.15)
    ds.tmean()
    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:

```
ensemble = nc.create_ensemble("../data/ensemble")
import multiprocessing
pool = multiprocessing.Pool(3)
for ff in ensemble:
    pool.apply_async(process_chain, [ff, ff.replace("ensemble", "new")])
pool.close()
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.

## 4.16 Examples

This tutorial runs through a number of example work flows.

### 4.16.1 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)
```

### 4.16.2 More to come....

## 4.17 Random Data Hacks

nctoolkit features a number of useful methods to tweak data.



### 4.17.1 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 multiple files with different fill values. You can do this using using `set_fill`:

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

### 4.17.2 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.

### 4.17.3 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.

#### 4.17.4 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-classic = "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")
```

#### 4.17.5 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 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()
```

#### 4.17.6 Removing leap days

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

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

#### 4.17.7 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"})
```

### 4.18 Global settings

nctoolkit let's you set global settings using options.

### 4.18.1 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)
```

### 4.18.2 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")
```

### 4.18.3 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')
```

### 4.18.4 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)
```

### 4.18.5 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 configuration 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.

## 4.19 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.

### 4.19.1 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)
```

### 4.19.2 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")
```

## 4.20 Troubleshooting

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.

### 4.20.1 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
```

### 4.20.2 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 dataset have the same horizontal grids.

### 4.20.3 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()
```

## 4.20.4 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.

## 4.21 API Reference

### 4.21.1 Session options

---

<code>options(\**kwargs)</code>	Define session options.
---------------------------------	-------------------------

---

#### nctoolkit.options

`nctoolkit.options(\**kwargs)`

Define session options. Set the options in the session. Available options are `thread_safe` and `lazy`. 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 `temp_dir = "/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.

**Parameters** **\*\*kwargs** – Define options using key, value pairs.

#### 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")
```

## 4.21.2 Opening/copying data

<code>open_data([x, checks])</code>	Read netCDF data as a DataSet object
<code>open_url([x, ftp_details, wait, file_stop])</code>	Read netCDF data from a url as a DataSet object
<code>open_thredds([x, wait, checks])</code>	Read thredds data as a DataSet object
<code>open_geotiff([x])</code>	Read geotiff and convert to nctoolkit dataset
<code>from_xarray(ds)</code>	Convert an xarray dataset to an nctoolkit dataset This will first save the xarray dataset as a temporary netCDF file.
<code>DataSet.copy(self)</code>	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
- **\*\*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("http://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("http://foo.nc")
```



### nctoolkit.open\_geotiff

nctoolkit.open\_geotiff(*x=[]*)

Read geotiff and convert to nctoolkit dataset

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

**Returns** *open\_data*

**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** :rtype: nctoolkit DataSet

## 4.21.3 Merging or analyzing multiple datasets

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

### 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.

#### Parameters

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

## 4.21.4 Adding and removing files to a dataset

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

### nctoolkit.DataSet.append

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

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 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")
```

**4.21.5 Accessing attributes**

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

**nctoolkit.DataSet.variables**

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

**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.

**nctoolkit.DataSet.times****property DataSet.times**

List times contained in a dataset

**nctoolkit.DataSet.years****property DataSet.years**

List years contained in a dataset

**nctoolkit.DataSet.months****property DataSet.months**

List months contained in a dataset

**nctoolkit.DataSet.levels****property DataSet.levels**

List levels contained in a dataset

**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.

**nctoolkit.DataSet.current****property DataSet.current**

The current file or files in the DataSet object

**nctoolkit.DataSet.history****property DataSet.history**

The history of operations on the DataSet

**nctoolkit.DataSet.start****property** DataSet.start

The starting file or files of the DataSet object

**nctoolkit.DataSet.calendar****property** DataSet.calendar

List calendars of dataset files

**nctoolkit.DataSet.ncformat****property** DataSet.ncformat

List formats of files contained in a dataset

**4.21.6 Plotting**


---

`DataSet.plot(self[, vars, autoscale, out, coast])`


---

**nctoolkit.DataSet.plot**

`DataSet.plot(self, vars=None, autoscale=True, out=None, coast=False, **kwargs)`

**4.21.7 Variable modification**

<code>DataSet.assign(self[, drop])</code>	Create new variables Existing columns that are re-assigned will be overwritten.
<code>DataSet.rename(self, newnames)</code>	Rename variables in a dataset
<code>DataSet.as_missing(self[, value])</code>	Change a range or individual value to missing.
<code>DataSet.missing_as(self[, value])</code>	Convert missing values to a constant
<code>DataSet.set_fill(self[, value])</code>	Set the fill value
<code>DataSet.sum_all(self[, drop])</code>	Calculate the sum of all variables for each time step

**nctoolkit.DataSet.assign**

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

Create new variables Existing columns that are re-assigned will be overwritten.

**Parameters**

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

## nctoolkit.DataSet.rename

`DataSet.rename(self, newnames)`

Rename variables in a dataset

**Parameters** **newnames** (*dict*) – Dictionary with key-value pairs being original and new variable names

## Examples

If you want to rename a variable x to y, 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.

## nctoolkit.DataSet.missing\_as

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

Convert missing values to a constant

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

## nctoolkit.DataSet.set\_fill

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

Set the fill value

**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.

**nctoolkit.DataSet.sum\_all**`DataSet.sum_all(self, drop=True)`

Calculate the sum of all variables for each time step

**Parameters** `drop` (*boolean*) – Do you want to keep variables?**4.21.8 netCDF file attribute modification**

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

**nctoolkit.DataSet.set\_longnames**`DataSet.set_longnames(self, name_dict=None)`

Set the long names of variables

**Parameters** `name_dict` (*dict*) – Dictionary with key, value pairs representing the variable names and their long names**nctoolkit.DataSet.set\_units**`DataSet.set_units(self, unit_dict=None)`

Set the units for variables

**Parameters** `unit_dict` (*dict*) – A dictionary where the key-value pairs are the variables and new units respectively.**4.21.9 Vertical/level methods**

<code>DataSet.top(self)</code>	Extract the top/surface level from a dataset This extracts the first vertical level from each file in a dataset.
<code>DataSet.bottom(self)</code>	Extract the bottom level from a dataset This extracts the bottom level from each netCDF file.
<code>DataSet.vertical_interp(self[, levels, ...])</code>	Vertically interpolate a dataset based on given vertical levels This is calculated for each time step and grid cell Note: This requires consistent vertical levels in space.
<code>DataSet.vertical_mean(self[, thickness, ...])</code>	Calculate the depth-averaged mean for each variable This is calculated for each time step and grid cell
<code>DataSet.vertical_min(self)</code>	Calculate the vertical minimum of variable values This is calculated for each time step and grid cell
<code>DataSet.vertical_max(self)</code>	Calculate the vertical maximum of variable values This is calculated for each time step and grid cell
<code>DataSet.vertical_range(self)</code>	Calculate the vertical range of variable values This is calculated for each time step and grid cell
<code>DataSet.vertical_sum(self)</code>	Calculate the vertical sum of variable values This is calculated for each time step and grid cell

continues on next page

Table 9 – continued from previous page

<code>DataSet.vertical_integration(self[, ...])</code>	Calculate the vertically integrated sum over the water column This calculates the sum of the variable multiplied by the cell thickness
<code>DataSet.vertical_cumsum(self)</code>	Calculate the vertical sum of variable values This is calculated for each time step and grid cell
<code>DataSet.invert_levels(self)</code>	Invert the levels of 3D variables This is calculated for each time step and grid cell
<code>DataSet.bottom_mask(self)</code>	Create a mask identifying the deepest cell without missing values.

## nctoolkit.DataSet.top

### DataSet.top(self)

Extract the top/surface level from a dataset This extracts the first vertical level from each file in a dataset.

#### Examples

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

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

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

## nctoolkit.DataSet.bottom

### DataSet.bottom(self)

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. 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)

Vertically interpolate a dataset based on given vertical levels This 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 True.
- **thickness** (*str or Dataset*) – This must be supplied if fixed is False, otherwise vertical thickness 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.

## 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)`

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*)

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*)

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*)

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*)

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*)

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

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

## nctoolkit.DataSet.vertical\_cumsum

**DataSet.vertical\_cumsum**(*self*)

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_cumsum()
```

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*)

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. At present this method only uses the first available variable from netCDF files, so it may not be suitable for all data

## 4.21.10 Rolling methods

<i>DataSet.rolling_mean</i> ( <i>self</i> [, window, align])	Calculate a rolling mean based on a window
<i>DataSet.rolling_min</i> ( <i>self</i> [, window, align])	Calculate a rolling minimum based on a window
<i>DataSet.rolling_max</i> ( <i>self</i> [, window, align])	Calculate a rolling maximum based on a window
<i>DataSet.rolling_sum</i> ( <i>self</i> [, window, align])	Calculate a rolling sum based on a window
<i>DataSet.rolling_range</i> ( <i>self</i> [, window, align])	Calculate a rolling range based on a window
<i>DataSet.rolling_stddev</i> ( <i>self</i> [, window, align])	Calculate a rolling standard deviation based on a window
<i>DataSet.rolling_var</i> ( <i>self</i> [, window, align])	Calculate a rolling variance based on a window

### nctoolkit.DataSet.rolling\_mean

**DataSet.rolling\_mean**(*self*, *window=None*, *align='right'*)

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')`

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')`

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')`

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')`

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')`

Calculate a rolling standard deviation 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 standard deviation with the standard deviation calculated over every 10 time steps, do the following:

```
>>> ds.rolling_sum(10)
```

### nctoolkit.DataSet.rolling\_var

`DataSet.rolling_var(self, window=None, align='right')`

Calculate a rolling variance 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 variance with the variance calculated over every 10 time steps, do the following:

```
>>> ds.rolling_sum(10)
```

## 4.21.11 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()
```

## 4.21.12 Cleaning functions

## 4.21.13 Ensemble creation

<code>create_ensemble([path, recursive])</code>	Generate an ensemble
---	----------------------

### nctoolkit.create\_ensemble

`nctoolkit.create_ensemble(path="", recursive=True)`

Generate an ensemble

#### 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** A list of files

**Return type** list

#### 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)
```

## 4.21.14 Arithmetic methods

<code>DataSet.abs(self)</code>	Method to get the absolute value of variables
<code>DataSet.add(self[, x, var])</code>	Add to a dataset This will add a constant, another dataset or a netCDF file to the dataset.
<code>DataSet.assign(self[, drop])</code>	Create new variables Existing columns that are re-assigned will be overwritten.
<code>DataSet.exp(self)</code>	Method to get the exponential of variables
<code>DataSet.log(self)</code>	Method to get the natural log of variables
<code>DataSet.log10(self)</code>	Method to get the base 10 log of variables
<code>DataSet.multiply(self[, x, var])</code>	Multiply a dataset This will multiply a dataset by a constant, another dataset or a netCDF file.
<code>DataSet.power(self[, x])</code>	Powers of variables in dataset

continues on next page



Table 14 – continued from previous page

<code>DataSet.sqrt(self)</code>	Method to get the square root of variables
<code>DataSet.square(self)</code>	Method to get the square of variables
<code>DataSet.subtract(self[, x, var])</code>	Subtract from a dataset This will subtract a constant, another dataset or a netCDF file from the dataset.
<code>DataSet.divide(self[, x, var])</code>	Divide the data This will divide the dataset by a constant, another dataset or a netCDF file.

## nctoolkit.DataSet.abs

`DataSet.abs(self)`

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 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
- -----
  - you wanted to add 10 to all variables in a dataset (*If*) –
  - would do the following (*you*) –
  - `ds.add(10) (>>>)` –
  - add the values in a dataset ds2 from a dataset ds1 (*To*) –
  - would do the following –
  - `ds1.add(ds2) (>>>)` –
  - in the datasets must match. Addition will occur in matching timesteps in ds1 and ds2. If there is only 1 timestep in ds2 (*Grids*) –
  - then –
  - data from that timestep will be added to the data in all ds1 time steps. (*the*) –
  - the data from another netCDF file will work in the same way (*Adding*) –

- `ds1.add("example.nc") (>>>) –`

### nctoolkit.DataSet.exp

`DataSet.exp(self)`

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)`

Method to get the natural log 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)`

Method to get the base 10 log 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 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)
```

To multiply the values in a dataset by the values of variables in dataset ds2, you would do the following:

```
>>> ds1.multiply(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)`

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)`

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)`

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.power()
```

## nctoolkit.DataSet.subtract

**DataSet.subtract**(*self*, *x=None*, *var=None*)

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)
```

To subtract the values in a dataset ds2 from those in a dataset ds1, you would do the following:

```
>>> ds1.subtract(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.

Subtracting of the data from another netCDF file will work in the same way:

```
>>> ds1.subtract("example.nc")
```

## nctoolkit.DataSet.divide

**DataSet.divide**(*self*, *x=None*, *var=None*)

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)
```

To divide values in a dataset by those in the dataset ds2 from a dataset ds1, you would do the following:

```
>>> ds1.divide(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.

Adding the data from another netCDF file will work in the same way:

```
>>> ds.divide("example.nc")
```

### 4.21.15 Ensemble statistics

<code>DataSet.ensemble_mean(self[, nco, ignore_time])</code>	Calculate an ensemble mean
<code>DataSet.ensemble_min(self[, nco, ignore_time])</code>	Calculate an ensemble min
<code>DataSet.ensemble_max(self[, nco, ignore_time])</code>	Calculate an ensemble maximum
<code>DataSet.ensemble_percentile(self[, p])</code>	Calculate an ensemble percentile This will calculate the percentles for each time step in the files.
<code>DataSet.ensemble_range(self)</code>	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.
<code>DataSet.ensemble_stdev(self)</code>	Calculate an ensemble standard deviation
<code>DataSet.ensemble_sum(self)</code>	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.
<code>DataSet.ensemble_var(self)</code>	Calculate an ensemble variance

#### nctoolkit.DataSet.ensemble\_mean

`DataSet.ensemble_mean(self, nco=False, ignore_time=False)`  
Calculate an ensemble mean

##### 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.

### nctoolkit.DataSet.ensemble\_min

`DataSet.ensemble_min(self, nco=False, ignore_time=False)`

Calculate an ensemble min

#### 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.

### nctoolkit.DataSet.ensemble\_max

`DataSet.ensemble_max(self, nco=False, ignore_time=False)`

Calculate an ensemble maximum

#### 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.

### nctoolkit.DataSet.ensemble\_percentile

`DataSet.ensemble_percentile(self, p=None)`

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.

**Parameters** *p* (*float or int*) – percentile to calculate.  $0 \leq p \leq 100$ .

### nctoolkit.DataSet.ensemble\_range

`DataSet.ensemble_range(self)`

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.

### nctoolkit.DataSet.ensemble\_stdev

`DataSet.ensemble_stdev(self)`

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.

**nctoolkit.DataSet.ensemble\_sum****DataSet.ensemble\_sum**(self)

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.

**nctoolkit.DataSet.ensemble\_var****DataSet.ensemble\_var**(self)

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.

**4.21.16 Subsetting operations**

<i>DataSet.subset</i> (self, <i>\**kwargs</i> )	A method for subsetting datasets to specific variables, years, longitudes etc.
<i>DataSet.crop</i> (self[, lon, lat, nco, nco_vars])	Crop to a rectangular longitude and latitude box
<i>DataSet.drop</i> (self, <i>\**kwargs</i> )	Remove variables This will remove stated variables from files in the dataset.

**nctoolkit.DataSet.subset****DataSet.subset**(self, *\\*\*kwargs*)

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.

**lat: list** The latitude range to select. This must be two variables, between -90 and 90.

**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 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)`

Remove variables 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
- **kwargs works as follows (Each) –**
- **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 February you would do the following:

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

## 4.21.17 Time-based methods

<code>DataSet.set_date(self[, year, month, day, ...])</code>	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.
<code>DataSet.set_day(self, x)</code>	Set the day for each time step in a dataset
<code>DataSet.shift(self, <i>kwargs</i>)</code>	Shift method.

### 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.

### 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

### nctoolkit.DataSet.shift

`DataSet.shift(self, *\kwargs)`

Shift method. A wrapper for shift\_days, shift\_hours 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)
```

#### 4.21.18 Interpolation, matching and resampling methods

<code>DataSet.regrid(self[, grid, method, recycle])</code>	Regrid a dataset to a target grid
<code>DataSet.to_latlon(self[, lon, lat, res, ...])</code>	Regrid a dataset to a regular latlon grid
<code>DataSet.match_points(self[, df, variables, ...])</code>	Match dataset to a spatiotemporal points dataframe
<code>DataSet.resample_grid(self[, factor])</code>	Resample the horizontal grid of a dataset
<code>DataSet.time_interp(self[, start, end, ...])</code>	Temporally interpolate variables based on date range and time resolution
<code>DataSet.timestep_interp(self[, steps])</code>	Temporally interpolate a dataset to given number of time steps between existing time steps
<code>DataSet.fill_na(self[, n])</code>	Fill missing values with a distance-weighted average.
<code>DataSet.box_mean(self[, x, y])</code>	Calculate the grid box mean for all variables This is performed for each time step.
<code>DataSet.box_max(self[, x, y])</code>	Calculate the grid box max for all variables This is performed for each time step.
<code>DataSet.box_min(self[, x, y])</code>	Calculate the grid box min for all variables This is performed for each time step.
<code>DataSet.box_sum(self[, x, y])</code>	Calculate the grid box sum for all variables This is performed for each time step.
<code>DataSet.box_range(self[, x, y])</code>	Calculate the grid box range for all variables This is performed for each time step.

#### nctoolkit.DataSet.regrid

`DataSet.regrid(self, grid=None, method='bil', recycle=False, \**kwargs)`

Regrid a dataset to a target grid

##### 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.
- **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*.

### nctoolkit.DataSet.to\_latlon

`DataSet.to_latlon(self, lon=None, lat=None, res=None, method='bil', recycle=False)`

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

### 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 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 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')`

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.

### nctoolkit.DataSet.timestep\_interp

`DataSet.timestep_interp(self, steps=None)`

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.

### nctoolkit.DataSet.fill\_na

`DataSet.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

**nctoolkit.DataSet.box\_mean**

`DataSet.box_mean(self, x=1, y=1)`

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

**nctoolkit.DataSet.box\_max**

`DataSet.box_max(self, x=1, y=1)`

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

**nctoolkit.DataSet.box\_min**

`DataSet.box_min(self, x=1, y=1)`

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

**nctoolkit.DataSet.box\_sum**

`DataSet.box_sum(self, x=1, y=1)`

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

**nctoolkit.DataSet.box\_range**

`DataSet.box_range(self, x=1, y=1)`

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

### 4.21.19 Masking methods

<code>DataSet.mask_box(self[, lon, lat])</code>	Mask a lon/lat box
---	--------------------

#### nctoolkit.DataSet.mask\_box

`DataSet.mask_box(self, lon=[- 180, 180], lat=[- 90, 90])`

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]

### 4.21.20 Anomaly methods

<code>DataSet.annual_anomaly(self[, baseline, ...])</code>	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.
<code>DataSet.monthly_anomaly(self[, baseline])</code>	Calculate monthly anomalies based on a baseline period The anomaly is derived by first calculating the climatological monthly mean for the given baseline period.

#### nctoolkit.DataSet.annual\_anomaly

`DataSet.annual_anomaly(self, baseline=None, metric='absolute', window=1)`

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)`

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 anomalies 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])
```

## 4.21.21 Statistical methods

<code>DataSet.tmean(self[, over, align])</code>	Calculate the temporal mean of all variables
<code>DataSet.tmin(self[, over, align])</code>	Calculate the temporal minimum of all variables
<code>DataSet.tmedian(self[, over, align])</code>	Calculate the temporal median of all variables
<code>DataSet.tpercentile(self[, p, over, align])</code>	Calculate the temporal percentile of all variables
<code>DataSet.tmax(self[, over, align])</code>	Calculate the temporal maximum of all variables
<code>DataSet.tsum(self[, over, align])</code>	Calculate the temporal sum of all variables
<code>DataSet.trange(self[, over, align])</code>	Calculate the temporal range of all variables
<code>DataSet.tstdev(self[, over, align])</code>	Calculate the temporal standard deviation of all variables
<code>DataSet.tcumsum(self[, align])</code>	Calculate the temporal cumulative sum of all variables
<code>DataSet.tvar(self[, over, align])</code>	Calculate the temporal variance of all variables

continues on next page



Table 21 – continued from previous page

<code>DataSet.cor_space(self[, var1, var2])</code>	Calculate the correlation correct between two variables in space This is calculated for each time step.
<code>DataSet.cor_time(self[, var1, var2])</code>	Calculate the correlation correct in time between two variables The correlation is calculated for each grid cell, ignoring missing values.
<code>DataSet.spatial_mean(self)</code>	Calculate the area weighted spatial mean for all variables This is performed for each time step.
<code>DataSet.spatial_min(self)</code>	Calculate the spatial minimum for all variables This is performed for each time step.
<code>DataSet.spatial_max(self)</code>	Calculate the spatial maximum for all variables This is performed for each time step.
<code>DataSet.spatial_percentile(self[, p])</code>	Calculate the spatial sum for all variables This is performed for each time step.
<code>DataSet.spatial_range(self)</code>	Calculate the spatial range for all variables This is performed for each time step.
<code>DataSet.spatial_sum(self[, by_area])</code>	Calculate the spatial sum for all variables This is performed for each time step.
<code>DataSet.spatial_stddev(self)</code>	Calculate the spatial range for all variables This is performed for each time step.
<code>DataSet.spatial_var(self)</code>	Calculate the spatial range for all variables This is performed for each time step.
<code>DataSet.centre(self[, by, by_area])</code>	Calculate the latitudinal or longitudinal centre for each year/month combination in files.
<code>DataSet.zonal_mean(self)</code>	Calculate the zonal mean for each year/month combination in files.
<code>DataSet.zonal_min(self)</code>	Calculate the zonal minimum for each year/month combination in files.
<code>DataSet.zonal_max(self)</code>	Calculate the zonal maximum for each year/month combination in files.
<code>DataSet.zonal_range(self)</code>	Calculate the zonal range for each year/month combination in files.
<code>DataSet.zonal_sum(self[, by_area])</code>	Calculate the zonal sum for each year/month combination in files.
<code>DataSet.meridional_mean(self)</code>	Calculate the meridional mean for each year/month combination in files.
<code>DataSet.meridional_min(self)</code>	Calculate the meridional minimum for each year/month combination in files.
<code>DataSet.meridional_max(self)</code>	Calculate the meridional maximum for each year/month combination in files.
<code>DataSet.meridional_range(self)</code>	Calculate the meridional range for each year/month combination in files.

## nctoolkit.DataSet.tmean

`DataSet.tmean(self, over='time', align='right')`  
Calculate the temporal mean of all variables

### Parameters

- **over** (*str* or *list*) – Time periods to average over. Options are ‘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 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 a dataset, do this:

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

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')`  
Calculate the temporal minimum of all variables

### Parameters

- **over** (*str* or *list*) – Time periods to average over. Options are ‘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 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')`

Calculate the temporal median of all variables

### Parameters

- **over** (*str or list*) – Time periods to average over. Options are ‘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 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')`

Calculate the temporal percentile of all variables

### Parameters

- **p** (*float or int*) – Percentile to calculate
- **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)
```

## nctoolkit.DataSet.tmax

`DataSet.tmax(self, over='time', align='right')`

Calculate the temporal maximum of all variables

### Parameters

- **over** (*str or list*) – Time periods to average over. Options are ‘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 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')`

Calculate the temporal sum of all variables

**Parameters = 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”

## nctoolkit.DataSet.trange

`DataSet.trange(self, over='time', align='right')`

Calculate the temporal range of all variables

### Parameters

- **over** (*str or list*) – Time periods to average over. Options are ‘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 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')`

Calculate the temporal standard deviation of all variables

### Parameters

- **over** (*str* or *list*) – Time periods to average over. Options are ‘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 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')`

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')`

Calculate the temporal variance of all variables

#### Parameters

- **over** (*str or list*) – Time periods to average over. Options are ‘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 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*)

Calculate the correlation correct between two variables in space This is calculated for each time step. The correlation coefficient is calculated using values in all grid cells, ignoring missing values.

### 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*)

Calculate the correlation correct in time between two variables The correlation is calculated for each grid cell, ignoring missing values.

### 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_space("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)`

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 that this calculation 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)`

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)`

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)`

Calculate the spatial sum for all variables This is performed for each time step.

**Parameters** *p* (*int* or *float*) – Percentile to calculate.  $0 \leq p \leq 100$ .

## 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)`

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)`

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)`

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\_var

`DataSet.spatial_var(self)`

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.centre

`DataSet.centre(self, by='latitude', by_area=False)`

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.

### nctoolkit.DataSet.zonal\_mean

`DataSet.zonal_mean(self)`

Calculate the zonal mean for each year/month combination in files. This applies to each file in an ensemble.

## 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*)

Calculate the zonal minimum for each year/month combination in files. This applies to each file in an ensemble.

**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*)

Calculate the zonal maximum for each year/month combination in files. This applies to each file in an ensemble.

**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*)

Calculate the zonal range for each year/month combination in files. This applies to each file in an ensemble.

**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*)

Calculate the zonal sum for each year/month combination in files. This applies to each file in an ensemble.

**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.meridional\_mean

`DataSet.meridional_mean(self)`

Calculate the meridional mean for each year/month combination in files. This applies to each file in an ensemble.

## Examples

If you want to calculate the meridional mean for a dataset, do the following:

```
>>> ds.meridional_mean()
```

## nctoolkit.DataSet.meridional\_min

`DataSet.meridional_min(self)`

Calculate the meridional minimum for each year/month combination in files. This applies to each file in an ensemble.

## Examples

If you want to calculate the meridional minimum for a dataset, do the following:

```
>>> ds.meridional_min()
```

## nctoolkit.DataSet.meridional\_max

`DataSet.meridional_max(self)`

Calculate the meridional maximum for each year/month combination in files. This applies to each file in an ensemble.

## Examples

If you want to calculate the meridional maximum for a dataset, do the following:

```
>>> ds.meridional_max()
```

### nctoolkit.DataSet.meridional\_range

`DataSet.meridional_range(self)`

Calculate the meridional range for each year/month combination in files. This applies to each file in an ensemble.

#### Examples

If you want to calculate the meridional range for a dataset, do the following:

```
>>> ds.meridional_max()
```

## 4.21.22 Merging methods

---

`DataSet.merge(self[, join, match, check])`

Merge a multi-file ensemble into a single file 2 methods are available.

---

### nctoolkit.DataSet.merge

`DataSet.merge(self, join='variables', match=['year', 'month', 'day'], check=True)`

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.

## 4.21.23 Splitting methods

---

`DataSet.split(self[, by])`

Split the dataset Each file in the ensemble will be separated into new files based on the splitting argument.

---

## nctoolkit.DataSet.split

`DataSet.split(self, by=None)`

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")
```

## 4.21.24 Output and formatting methods

<code>DataSet.to_nc(self, out[, zip, overwrite])</code>	Save a dataset to a named file This will only work with single file datasets.
<code>DataSet.to_xarray(self[, decode_times])</code>	Open a dataset as an xarray object
<code>DataSet.to_dataframe(self[, decode_times])</code>	Open a dataset as a pandas data frame
<code>DataSet.zip(self)</code>	Zip the dataset This will compress the files within the dataset.
<code>DataSet.format(self[, ext])</code>	Zip the dataset This will compress the files within the dataset.

## nctoolkit.DataSet.to\_nc

`DataSet.to_nc(self, out, zip=True, overwrite=False, \**kwargs)`

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)`

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)`

Open a dataset as 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

### nctoolkit.DataSet.zip

`DataSet.zip(self)`

Zip the dataset This will compress the files within the dataset. This works lazily.

#### Examples

If you want to zip the files in a dataset, do the following:

```
>>> ds.zip()
```

This will occur lazily, so will only occur after everything has been evaluated.

### nctoolkit.DataSet.format

`DataSet.format(self, ext=None)`

Zip the 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-classic = nc4c netCDF version 5 (64-bit data) = nc5

## 4.21.25 Miscellaneous methods

<code>DataSet.na_count(self[, over, align])</code>	Calculate the number of missing values
<code>DataSet.na_frac(self[, over, align])</code>	Calculate the number of missing values
<code>DataSet.distribute(self[, m, n])</code>	Split the dataset into multiple evenly sized horizontal and vertical new files
<code>DataSet.collect(self)</code>	Collect a dataset that has been split using distribute
<code>DataSet.cell_area(self[, join])</code>	Calculate the area of grid cells.
<code>DataSet.first_above(self[, x])</code>	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.
<code>DataSet.first_below(self[, x])</code>	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.

continues on next page

Table 25 – continued from previous page

<code>DataSet.last_above(self[, x])</code>	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.
<code>DataSet.last_below(self[, x])</code>	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.
<code>DataSet.cdo_command(self[, command, ensemble])</code>	Apply a cdo command
<code>DataSet.nco_command(self[, command, ensemble])</code>	Apply an nco command
<code>DataSet.compare(self[, expression])</code>	Compare all variables to a constant
<code>DataSet.gt(self, x)</code>	Method to calculate if variable in dataset is greater than that in another file or dataset This currently only works with single file datasets
<code>DataSet.lt(self, x)</code>	Method to calculate if variable in dataset is less than that in another file or dataset This currently only works with single file datasets
<code>DataSet.reduce_dims(self)</code>	Reduce dimensions of data This will remove any dimensions with only one value.
<code>DataSet.reduce_grid(self[, mask])</code>	Reduce the dataset to non-zero locations in a mask
<code>DataSet.set_precision(self, x)</code>	Set the precision in a dataset
<code>DataSet.check(self)</code>	Check contents of files for common data problems.
<code>DataSet.is_corrupt(self)</code>	Check if files are corrupt
<code>DataSet.fix_nemo_ersem_grid(self)</code>	A quick hack to change the grid file in North West European shelf Nemo grids.
<code>DataSet.set_gridtype(self, grid)</code>	Set the grid type.
<code>DataSet.surface_mask(self)</code>	Create a mask identifying the shallowest cell without missing values.
<code>DataSet.strip_variables(self[, vars])</code>	Remove any variables, such as bnds etc., from variables.
<code>DataSet.no_leaps(self)</code>	Remove leap years.
<code>DataSet.as_double(self, x)</code>	Set a variable/dimension to double This is mostly useful for cases when time is stored as an int, but you need a double
<code>DataSet.as_type(self, x)</code>	Set a variable/dimension to double This is mostly useful for cases when time is stored as an int, but you need a double
<code>DataSet.reset(self)</code>	Simple method to fully reset a dataset

**nctoolkit.DataSet.na\_count**

`DataSet.na_count(self, over='time', align='right')`

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”

**nctoolkit.DataSet.na\_frac**

**DataSet.na\_frac**(*self*, *over*='time', *align*='right')

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”

**nctoolkit.DataSet.distribute**

**DataSet.distribute**(*self*, *m*=1, *n*=1)

Split the dataset into multiple evenly sized horizontal and vertical new files

**Parameters**

- **m** (*int*) – Number of rows
- **n** (*int*) – Number of columns

**nctoolkit.DataSet.collect**

**DataSet.collect**(*self*)

Collect a dataset that has been split using distribute

**nctoolkit.DataSet.cell\_area**

**DataSet.cell\_area**(*self*, *join*=True)

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)`

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)`

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*)

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*)

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)`

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 command?

### 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.

### 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 will be calculated for each time step.

If you wanted to identify grid cells with negative values, you would do this

```
>>> ds.compare("<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 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 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'.
- `month` (*int*) – The month
- `day` (*int*) – The day
- `base_year` (*int*) – The base year for time creation in the netCDF. Defaults to 1900.

**nctoolkit.DataSet.check**

**DataSet.check**(*self*)

Check contents of files for common data problems.

**nctoolkit.DataSet.is\_corrupt**

**DataSet.is\_corrupt**(*self*)

Check if files are 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*)

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

**nctoolkit.DataSet.strip\_variables**

**DataSet.strip\_variables**(*self*, *vars=None*)

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



**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

**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'.

**nctoolkit.DataSet.reset****DataSet.reset**(*self*)

Simple method to fully reset a dataset

**4.21.26 Ecological methods***DataSet.phenology*(*self*[, *var*, *metric*, *p*])

Calculate phenologies from a dataset Each file in an ensemble must only cover a single year, and ideally have all days.

**nctoolkit.DataSet.phenology****DataSet.phenology**(*self*, *var=None*, *metric=None*, *p=None*)

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.

## 4.22 Package info

This package was created by Robert Wilson at Plymouth Marine Laboratory (PML).

### 4.22.1 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.

### 4.22.2 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](mailto:rwi@pml.ac.uk).

### 4.22.3 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 an issue at the package's GitHub page or contact me.

## 4.23 Cheat sheet

A cheat sheet providing a quick 2-page overview of nctoolkit is available [here](#).

## A

abs() (*nctoolkit.DataSet* method), 77  
 add() (*nctoolkit.DataSet* method), 77  
 annual\_anomaly() (*nctoolkit.DataSet* method), 91  
 append() (*nctoolkit.DataSet* method), 62  
 as\_double() (*nctoolkit.DataSet* method), 117  
 as\_missing() (*nctoolkit.DataSet* method), 66  
 as\_type() (*nctoolkit.DataSet* method), 117  
 assign() (*nctoolkit.DataSet* method), 65

## B

bottom() (*nctoolkit.DataSet* method), 68  
 bottom\_mask() (*nctoolkit.DataSet* method), 72  
 box\_max() (*nctoolkit.DataSet* method), 90  
 box\_mean() (*nctoolkit.DataSet* method), 90  
 box\_min() (*nctoolkit.DataSet* method), 90  
 box\_range() (*nctoolkit.DataSet* method), 90  
 box\_sum() (*nctoolkit.DataSet* method), 90

## C

calendar (*nctoolkit.DataSet* property), 65  
 cdo\_command() (*nctoolkit.DataSet* method), 114  
 cell\_area() (*nctoolkit.DataSet* method), 111  
 centre() (*nctoolkit.DataSet* method), 103  
 check() (*nctoolkit.DataSet* method), 116  
 collect() (*nctoolkit.DataSet* method), 111  
 compare() (*nctoolkit.DataSet* method), 114  
 contents (*nctoolkit.DataSet* property), 64  
 copy() (*nctoolkit.DataSet* method), 61  
 cor\_space() (*in module nctoolkit*), 62  
 cor\_space() (*nctoolkit.DataSet* method), 100  
 cor\_time() (*in module nctoolkit*), 62  
 cor\_time() (*nctoolkit.DataSet* method), 100  
 create\_ensemble() (*in module nctoolkit*), 76  
 crop() (*nctoolkit.DataSet* method), 84  
 current (*nctoolkit.DataSet* property), 64

## D

distribute() (*nctoolkit.DataSet* method), 111  
 divide() (*nctoolkit.DataSet* method), 80  
 drop() (*nctoolkit.DataSet* method), 85

## E

ensemble\_max() (*nctoolkit.DataSet* method), 82  
 ensemble\_mean() (*nctoolkit.DataSet* method), 81  
 ensemble\_min() (*nctoolkit.DataSet* method), 82  
 ensemble\_percentile() (*nctoolkit.DataSet* method), 82  
 ensemble\_range() (*nctoolkit.DataSet* method), 82  
 ensemble\_stdev() (*nctoolkit.DataSet* method), 82  
 ensemble\_sum() (*nctoolkit.DataSet* method), 83  
 ensemble\_var() (*nctoolkit.DataSet* method), 83  
 exp() (*nctoolkit.DataSet* method), 78

## F

fill\_na() (*nctoolkit.DataSet* method), 89  
 first\_above() (*nctoolkit.DataSet* method), 112  
 first\_below() (*nctoolkit.DataSet* method), 112  
 fix\_nemo\_ersem\_grid() (*nctoolkit.DataSet* method), 116  
 format() (*nctoolkit.DataSet* method), 109  
 from\_xarray() (*in module nctoolkit*), 61

## G

gt() (*nctoolkit.DataSet* method), 114

## H

history (*nctoolkit.DataSet* property), 64

## I

invert\_levels() (*nctoolkit.DataSet* method), 71  
 is\_corrupt() (*nctoolkit.DataSet* method), 116

## L

last\_above() (*nctoolkit.DataSet* method), 113  
 last\_below() (*nctoolkit.DataSet* method), 113  
 levels (*nctoolkit.DataSet* property), 64  
 log() (*nctoolkit.DataSet* method), 78  
 log10() (*nctoolkit.DataSet* method), 78  
 lt() (*nctoolkit.DataSet* method), 115

## M

mask\_box() (*nctoolkit.DataSet* method), 91

`match_points()` (*nctoolkit.DataSet* method), 88  
`merge()` (*in module nctoolkit*), 61  
`merge()` (*nctoolkit.DataSet* method), 106  
`meridional_max()` (*nctoolkit.DataSet* method), 105  
`meridional_mean()` (*nctoolkit.DataSet* method), 105  
`meridional_min()` (*nctoolkit.DataSet* method), 105  
`meridional_range()` (*nctoolkit.DataSet* method), 106  
`missing_as()` (*nctoolkit.DataSet* method), 66  
`monthly_anomaly()` (*nctoolkit.DataSet* method), 92  
`months` (*nctoolkit.DataSet* property), 64  
`multiply()` (*nctoolkit.DataSet* method), 78

## N

`na_count()` (*nctoolkit.DataSet* method), 110  
`na_frac()` (*nctoolkit.DataSet* method), 111  
`ncformat` (*nctoolkit.DataSet* property), 65  
`nco_command()` (*nctoolkit.DataSet* method), 114  
`no_leaps()` (*nctoolkit.DataSet* method), 116

## O

`open_data()` (*in module nctoolkit*), 59  
`open_geotiff()` (*in module nctoolkit*), 61  
`open_thredds()` (*in module nctoolkit*), 60  
`open_url()` (*in module nctoolkit*), 60  
`options()` (*in module nctoolkit*), 58

## P

`phenology()` (*nctoolkit.DataSet* method), 117  
`plot()` (*nctoolkit.DataSet* method), 65  
`power()` (*nctoolkit.DataSet* method), 79

## R

`reduce_dims()` (*nctoolkit.DataSet* method), 115  
`reduce_grid()` (*nctoolkit.DataSet* method), 115  
`regrid()` (*nctoolkit.DataSet* method), 87  
`remove()` (*nctoolkit.DataSet* method), 63  
`rename()` (*nctoolkit.DataSet* method), 66  
`resample_grid()` (*nctoolkit.DataSet* method), 89  
`reset()` (*nctoolkit.DataSet* method), 117  
`rolling_max()` (*nctoolkit.DataSet* method), 73  
`rolling_mean()` (*nctoolkit.DataSet* method), 72  
`rolling_min()` (*nctoolkit.DataSet* method), 73  
`rolling_range()` (*nctoolkit.DataSet* method), 74  
`rolling_stddev()` (*nctoolkit.DataSet* method), 74  
`rolling_sum()` (*nctoolkit.DataSet* method), 74  
`rolling_var()` (*nctoolkit.DataSet* method), 75  
`run()` (*nctoolkit.DataSet* method), 75

## S

`set_date()` (*nctoolkit.DataSet* method), 86  
`set_day()` (*nctoolkit.DataSet* method), 86  
`set_fill()` (*nctoolkit.DataSet* method), 66  
`set_gridtype()` (*nctoolkit.DataSet* method), 116

`set_longnames()` (*nctoolkit.DataSet* method), 67  
`set_precision()` (*nctoolkit.DataSet* method), 115  
`set_units()` (*nctoolkit.DataSet* method), 67  
`shift()` (*nctoolkit.DataSet* method), 86  
`size` (*nctoolkit.DataSet* property), 64  
`spatial_max()` (*nctoolkit.DataSet* method), 101  
`spatial_mean()` (*nctoolkit.DataSet* method), 101  
`spatial_min()` (*nctoolkit.DataSet* method), 101  
`spatial_percentile()` (*nctoolkit.DataSet* method), 101  
`spatial_range()` (*nctoolkit.DataSet* method), 102  
`spatial_stddev()` (*nctoolkit.DataSet* method), 102  
`spatial_sum()` (*nctoolkit.DataSet* method), 102  
`spatial_var()` (*nctoolkit.DataSet* method), 103  
`split()` (*nctoolkit.DataSet* method), 107  
`sqrt()` (*nctoolkit.DataSet* method), 79  
`square()` (*nctoolkit.DataSet* method), 79  
`start` (*nctoolkit.DataSet* property), 65  
`strip_variables()` (*nctoolkit.DataSet* method), 116  
`subset()` (*nctoolkit.DataSet* method), 83  
`subtract()` (*nctoolkit.DataSet* method), 80  
`sum_all()` (*nctoolkit.DataSet* method), 67  
`surface_mask()` (*nctoolkit.DataSet* method), 116

## T

`tcumsum()` (*nctoolkit.DataSet* method), 99  
`time_interp()` (*nctoolkit.DataSet* method), 89  
`times` (*nctoolkit.DataSet* property), 64  
`timestep_interp()` (*nctoolkit.DataSet* method), 89  
`tmax()` (*nctoolkit.DataSet* method), 96  
`tmean()` (*nctoolkit.DataSet* method), 94  
`tmedian()` (*nctoolkit.DataSet* method), 95  
`tmin()` (*nctoolkit.DataSet* method), 94  
`to_dataframe()` (*nctoolkit.DataSet* method), 109  
`to_latlon()` (*nctoolkit.DataSet* method), 88  
`to_nc()` (*nctoolkit.DataSet* method), 107  
`to_xarray()` (*nctoolkit.DataSet* method), 108  
`top()` (*nctoolkit.DataSet* method), 68  
`tpercentile()` (*nctoolkit.DataSet* method), 96  
`trange()` (*nctoolkit.DataSet* method), 97  
`tstddev()` (*nctoolkit.DataSet* method), 98  
`tsum()` (*nctoolkit.DataSet* method), 97  
`tvar()` (*nctoolkit.DataSet* method), 99

## V

`variables` (*nctoolkit.DataSet* property), 63  
`vertical_cumsum()` (*nctoolkit.DataSet* method), 71  
`vertical_integration()` (*nctoolkit.DataSet* method), 71  
`vertical_interp()` (*nctoolkit.DataSet* method), 68  
`vertical_max()` (*nctoolkit.DataSet* method), 70  
`vertical_mean()` (*nctoolkit.DataSet* method), 69  
`vertical_min()` (*nctoolkit.DataSet* method), 70  
`vertical_range()` (*nctoolkit.DataSet* method), 70

`vertical_sum()` (*nctoolkit.DataSet* method), [70](#)

## Y

`years` (*nctoolkit.DataSet* property), [64](#)

## Z

`zip()` (*nctoolkit.DataSet* method), [109](#)

`zonal_max()` (*nctoolkit.DataSet* method), [104](#)

`zonal_mean()` (*nctoolkit.DataSet* method), [103](#)

`zonal_min()` (*nctoolkit.DataSet* method), [104](#)

`zonal_range()` (*nctoolkit.DataSet* method), [104](#)

`zonal_sum()` (*nctoolkit.DataSet* method), [104](#)