nctoolkit

Robert Wilson

Jul 07, 2021

QUICK OVERVIEW

nctoolkit is a comprehensive Python package for analyzing netCDF data on Linux and macOS.

Core abilities 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

CHAPTER ONE

INSTALLATION

1.1 How to install nctoolkit

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 here 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
```

1.2 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

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

CHAPTER

TWO

INTRODUCTION TO NCTOOLKIT

nctoolkit is a multi-purpose tool for analyzing and post-processing netCDF files. We will see what it is capable of by carrying out an exploratory analysis of sea surface temperature since 1850.

We will use the temperature data set COBE2 from the National Oceanic and Atmospheric Administration. This is a global dataset of sea surface temperature at a horizontal resolution of 1 degree for every month since 1850.

It is best to import nctoolkit as follows:

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

We will set the file paths as follows:

```
[3]: ff = "sst.mon.mean.nc"
```

nctoolkit works with datasets. These contain either a single file or a list of files that we will work with. We can create using the temperature file, as follows:

```
[5]: ds = nc.open_data(ff)
```

We can access easily access the attributes of the dataset. For example, if we wanted to find out the number of years in the dataset, we can do this:

```
[6]: [min(ds.years), max(ds.years)]
```

```
[6]: [1850, 2019]
```

All years from 1850 to 2019 are available. We could find out the variables available like so:

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

```
[7]: ['sst']
```

Now, if we want to do anything with a dataset, we need to use nctoolkit's many methods. Let's say we want to map mean temperature for the year 2016. We could do that as follows:

```
[9]: ds = nc.open_data(ff)
     ds.select(year = 2000)
     ds.tmean()
     ds.plot()
```

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

This was carried out in 3 clear steps. First, we selected the year 2000. Second, we calculated the temporal mean for that year. And we then plotted the result.

We might want to do something more interesting. We have a dataset of sea surface temperature. How much has the ocean warmed over this time? We can calculate that as follows:

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

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

Here we did the calculation in two steps. First we used `tmean` to calculate the annual mean since 1850 for each grid cell. We use the `year` keyword to tell nctoolkit that the mean should calculated each year. We then use `spatial_mean` to calculate the spatial mean.

Now, we might want to map how much the oceans have warmed over the last century. We could do this as follows:

```
[11]: ds_start = nc.open_data(ff)
      ds_start.select(years = range(1900, 1920))
      ds_start.tmean()

      ds_increase = nc.open_data(ff)
      ds_increase.select(years = range(2000, 2020))
      ds_increase.tmean()
      ds_increase.subtract(ds_start)
```

First, we created a dataset which gives the mean temperature between 1900 and 1919. We then create a second dataset, which initially is the mean temperature between 2000 and 2019. We then subtract the 1900-19 temperature from this dataset. We can now plot the results:

```
[12]: ds_increase.plot()
```

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

You can see that most of the world's oceans have warmed, but some have warmed more than others.

We might want to know how much oceans have warmed or cooled relative to the rest of the planet. We can do this using the `assign` method:

ds_increase.assign(sst = lambda x: x.sst - spatial_mean(x.sst)) ds_increase.plot()

Areas in the red warmed more than the global average.

2.1 Under the hood

Let's revisit the first code example to see how nctoolkit works behind the scenes:

```
[13]: ds = nc.open_data(ff)
      ds.select(year = 2000)
      ds.tmean()
```

The plotting part has been removed. Each dataset is made of up of files. We can see what they are as follows:

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

```
[14]: ['sst.mon.mean.nc']
```

You can see that this is just the file we started with. What's going on? The answer: nctoolkit works lazily. All calculations are carried out when the user says to, or when they have to be. To force calculations to be carried out, we use `run`. The `plot` method will, of course, for everything to be evaluated before plotting.

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

We can now see that the file in the dataset has changed:

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

```
[16]: ['/tmp/nctoolkitcrxkamxznctoolkittmp9t70e7fz.nc']
```

This is now a new temporary file. Under the hood, nctoolkit uses Climate Data Operators CDO. CDO is a powerful and ultra-efficient system for working with netCDF files. nctoolkit requires no knowledge of CDO, but if you want to understand it further you can read their excellent user guide.

We can see the CDO commands by access the history attribute:

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

```
[17]: ['cdo -L -timmean -selyear,2000 sst.mon.mean.nc /tmp/
      ↪nctoolkitcrxkamxznctoolkittmp9t70e7fz.nc']
```

You can see that 2 nctoolkit methods have been converted into one CDO call.

And don't worry, nctoolkit will automatically remove all of the temporary files once they are no longer needed.

Click on the tabs on the left to find out what nctoolkit is capable of

CHAPTER THREE

NEWS

3.1 Release of v0.3.6

Version 0.3.5 will be released in July 2021. This will be a minor release.

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

3.2 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`.

3.3 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`.

3.4 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.

3.5 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.

3.6 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.

CHAPTER

FOUR

DATASETS

nctoolkit works with what it calls datasets. Each dataset is made up of or more netCDF files.

4.1 Opening datasets

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

If the data you want to analyze that 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. So if you wanted to open all of the files in a folder called data as a dataset, you could do the following:

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

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 user `open_thredds`. The file paths should end with .nc.

4.2 Visualization of datasets

You can visualize the contents of a dataset using the `plot` method. Below, we will plot temperature for January and the North Atlantic:

```
ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE/data.mon.ltm.1981-
↪2010.nc")
ds.plot()
```

Please note there may be some issues due to bugs in nctoolkit's dependencies that cause problems plotting some data types. If data does not plot, raise an issue here.

4.3 Modifying datasets and lazy evaluation

nctoolkit works by performing operations and then saving the results as either a temporary file or in a file specified by the user. We can illustrate this with the following code. This will select the first time step from a file available over thredds and will plot the results.

```
ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE/data.mon.ltm.1981-
↪2010.nc")
ds.select(time = 0)
ds.plot()
```

You will notice, once this is done, that the file associated with the dataset is now a temporary file.

```
ds.current
```

This will happen each time nctoolkit carries out an operation. This is potentially an invitation to slow-running code. You do not want to be constantly reading and writing data. Ideally, you want a processing chain which minimizes IO. nctoolkit enables this by allowing method chaining, thanks to the method chaining of its computational back-end CDO.

Let's look at this chain of code:

```
ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE/data.mon.ltm.1981-
↪2010.nc")
ds.assign(sst = lambda x: x.sst + 273.15)
ds.select(months = 1)
ds.crop(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.

The quickest way to evaluate everything using `run`. The code above would become:

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

Evaluation is, to use the technical term, lazy within nctoolkit. It only evaluates things until it needs to or is forced to.

This allows us to create efficient processing chain where we read the input file and write to the output file with no intermediate file writing. If, in the example above, we wanted to save the output file, we could do this:

```
ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE/data.mon.ltm.1981-
↪2010.nc")
ds.select(months = 1)
ds.crop(lon = [-80, 20], lat = [30, 70])
ds.spatial_mean()
ds.to_nc("foo.nc")
```

4.4 List-like behaviour of datasets

If you want to view the files within a dataset view the `current` attribute.

This is a list that gives the file(s) within the dataset. To make processing these files easier nctoolkit features a number of methods similar to lists.

First, datasets are iterable. So, you can loop through each element of a dataset as follows:

You can find out how many files are in a dataset, using `len`:

You can add a new file to a dataset using `append`:

This method also let you add the files from another dataset.

Similarly, you can remove files from a dataset using `remove`:

In line with typical list behaviours, you can also create empty datasets as follows:

This is particularly useful if you need to create an ensemble based on multiple files that need significant processing before being added to the dataset.

4.5 Dataset attributes

We can find out key information about a dataset using its attributes.

If we want to know the variables available in a dataset called ds, we would do:

```
ds.variables
```

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

```
ds.levels
```

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

```
ds.times
```

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

```
ds.months
```

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

```
ds.years
```

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
```

CHAPTER

FIVE

CHEAT SHEET

A cheat sheet providing a quick 2-page overview of nctoolkit is available here.

CHAPTER SIX

PLOTTING

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

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

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

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

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

```
nctoolkit is using CDO version 1.9.9
```

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

```
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
```

(continues on next page)

(continued from previous page)

```
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
```

[1]:
```
:DynamicMap   []
   :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.select(year = 2000)
ds.tmean()
ds.zonal_mean()
ds.plot()
```

```
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
```

[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()
```

```
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
```

[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()
```

```
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
```

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

6.1 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 pallette when there are positives and negatives and a viridis palette otherwise.

CHAPTER

SEVEN

IMPORTING AND EXPORTING DATA

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

7.1 Opening single files and ensembles

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

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

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

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

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

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

7.2 Opening files from urls/ftp

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

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

7.3 Opening data available over thredds servers or OPeNDAP

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

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

7.4 Exporting datasets

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

7.5 Save as a netCDF

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

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

7.6 Convert to xarray Dataset

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()
```

7.7 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()
```

CHAPTER

EIGHT

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"])
```

8.1 Calculating rolling averages

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

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

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

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

8.2 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.

8.3 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")
```

8.4 Calculating climatologies

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

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

8.5 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.

CHAPTER

NINE

SUBSETTING DATA

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

9.1 Selecting variables

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

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

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

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

9.2 Selecting years

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

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

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

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

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

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

9.3 Selecting months

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

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

9.4 Selecting seasons

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

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

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

9.6 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])
```

CHAPTER

TEN

MANIPULATING VARIABLES

10.1 Creating new variables

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

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

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

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

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

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

This results in only one variable.

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

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

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

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

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

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

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

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

10.2 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))
```

10.3 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)
```

10.4 Functions that work with nctoolkit variables

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

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

10.5 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.

10.6 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")
```

CHAPTER ELEVEN

INTERPOLATION

nctoolkit features built in methods for horizontal and vertical interpolation.

11.1 Interpolating to a set of coordinates

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

```
import nctoolkit as nc
import pandas as pd
ds = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.mean.nc")
ds.select(timestep = range(0, 12))
coords = pd.DataFrame({"lon":[-30], "lat":[50]})
ds.regrid(coords)
```

11.1.1 Interpolating to a regular latlon 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:

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

11.1.2 Interpolating to another dataset's grid

If we are working with two datasets and want to put them on a common grid, we can interpolate one onto the other's grid. We can illustate this with a dataset of global sea surface temperature. Let's start by regridding the first timestep in this dataset to a regular latlon grid covering the North Atlantic.

```
ds1 = nc.open_thredds("https://psl.noaa.gov/thredds/dodsC/Datasets/COBE2/sst.mon.mean.nc
↪")
ds1.select(timestep = 0)
ds1.to_latlon(lon = [-79.5, 79.5], lat = [-0.75, 89.75], res = [1, 0.5])
```

We can then use this new dataset as the target grid in `regrid`. So

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

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.

11.1.3 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`.

The example below illustrates this. First, we regrid a global dataset to a regular latlon grid covering the North Atlantic, setting the recycle argument to True.

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

We can then use the grid from data for regridding:

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

This, of course, requires that the grids in the datasets are consistent. If you want to access the weights and grid files generated, you can do the following:

These files are deleted either when `data` is deleted or when the Python session is existed.

11.1.4 Resampling

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

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

11.1.5 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 ocean temperatures from NOAA's World Ocean Atlas for January to a depth of 500 metres. The `vertical_interp` method requires a `levels` argument, which is sea-depth in this case.

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

CHAPTER

TWELVE

ENSEMBLE METHODS

12.1 Merging files with different variables

This notebook will outline some general methods for doing comparisons of multiple files. We will work with two different sea surface temperature data sets from NOAA and the Met Office Hadley Centre.

```
[1]: import nctoolkit as nc
     import pandas as pd
     import xarray as xr
     import numpy as np
```

```
nctoolkit is using CDO version 1.9.8
```

Let's start by downloading the files using wget. Uncomment the code below to do this (note: you will need to extract the HadISST dataset):

```
[2]: # ! wget ftp://ftp.cdc.noaa.gov/Datasets/COBE2/sst.mon.mean.nc
     # ! wget https://www.metoffice.gov.uk/hadobs/hadisst/data/HadISST_sst.nc.gz
```

The first step is to get the data. We will start by creating two separate datasets for each file.

```
[3]: sst_noaa = nc.open_data("sst.mon.mean.nc")
     sst_hadley = nc.open_data("HadISST_sst.nc")
```

We can see that both variables have sea surface temperature labelled as sst. So we will need to change that.

```
[4]: sst_noaa.variables
```

```
[4]: ['sst']
```

```
[5]: sst_hadley.variables
```

```
[5]: ['sst', 'time_bnds']
```

```
[6]: sst_noaa.rename({"sst":"noaa"})
     sst_hadley.rename({"sst":"hadley"})
```

The data sets also cover different time periods, and only have overlapping between 1870 and 2018. so we will need to select those years

```
[7]: sst_noaa.select(years = range(1870, 2019))
     sst_hadley.select(years = range(1870, 2019))
```

We also have a problem in that there are two horizontal grids in the Hadley Centre file. We can solve this by selecting the sst variable only

```
[8]: sst_hadley.select(variables = "hadley")
```

At this point, the datasets have the same number of time steps and months covered. However, the grids are still a bit different. So we want to unify them by regridding one dataset on to the other's grid. This can be done using regrid, or any grid of your choosing.

```
[9]: sst_noaa.regrid(grid = sst_hadley)
```

We now have two separate datasets. Let's create a new dataset that has both of them, and then merge them. When doing this we need to make sure nas are treated properly. In this case Hadley Centre values not being NAs as they should be, so we need to fix that. The merge method also requires a strict matching criteria for the dates in the merging files. In this case the Hadley Centre and NOAA data sets both give monthly means, but use a different day of the month. So we will set match to ["year", "month"] this will ensure there are no mis-matches

```
[10]: all_sst = nc.merge(sst_noaa, sst_hadley, match = ["year", "month"])
      all_sst.set_missing([-9000, - 900])
```

Let's work out what the global mean SST was over the time period. Note that this will not be totally accurate as there are some missing values here and there that might bias things.

```
[11]: all_sst.spatial_mean()
      all_sst.tmean("year")
      all_sst.rolling_mean(10)
```

```
[12]: all_sst.plot("noaa")
```

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

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

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

We can also work out the difference between the two. Here we wil work out the monthly bias per cell. Then calculate the mean global difference per year, and then calculate a rolling 10 year mean.

```
[13]: all_sst = nc.open_data([sst_noaa.current, sst_hadley.current])
      all_sst.merge(match = ["year", "month"])
      all_sst.transmute({"bias":"hadley-noaa"})
      all_sst.set_missing([-9000, - 900])
      all_sst.spatial_mean()
      all_sst.tmean("year")
      all_sst.rolling_mean(10)
      all_sst.plot("bias")
```

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

You can see that there is a notable difference at the start of the time series.

12.2 Merging files with different times

TBC

12.3 Ensemble averaging

TBC

CHAPTER

THIRTEEN

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.

13.1 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.

13.2 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.

CHAPTER

FOURTEEN

EXAMPLES

This tutorial runs through a number of example work flows.

14.1 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 CDO version 1.9.9
```

[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

```
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
```

(continues on next page)

(continued from previous page)

```
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
```

[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")
```

```
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
```

[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")
```

```
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
```

[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")
```

```
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
WARNING:param.ParameterizedMetaclass: Use method 'params' via param namespace
```

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

14.2 More to come....

CHAPTER

FIFTEEN

RANDOM DATA HACKS

nctoolkit features a number of useful methods to tweak data.

15.1 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.

15.2 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.

15.3 Changing the format of the netCDF files in a dataset

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

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

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

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

15.4 Getting rid of dimensions with only one value

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

```
ds.reduce_dims()
```

CHAPTER

SIXTEEN

GLOBAL SETTINGS

nctoolkit let's you set global settings using options.

The most important and recommended to update is to set evaluation to lazy. This can be done as follows:

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

This means that commands will only be evaluated when either request them to be or they need to be.

For example, in the code below the 3 specified commands will only be calculated after it is told to `run`. This cuts down on IO, and can result in significant improvements in run time. At present lazy defaults to False, but this may change in a future release of nctoolkit.

```
nc.options(lazy = True)
data.tmean()
data.crop(lat = [0,90])
data.spatial_mean()
data.run()
```

If you are working with ensembles, you may want 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. By default cores = 1. Most methods can run in parallel when working with multi-file datasets.

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

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")
```

16.1 Setting global settings using a configuration file

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

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

```
lazy : True

cores : 6
```

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

CHAPTER

SEVENTEEN

API REFERENCE

17.1 Session options

options(**kwargs)	Define session options.

17.1.1 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 = True if you want methods to evaluate lazy by default. 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.

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")
```

17.2 Opening/copying data

open_data([x, checks])	Read netCDF data as a DataSet object
open_url([x, ftp_details, wait, file_stop])	Read netCDF data from a url as a DataSet object
open_thredds([x, wait, checks])	Read thredds data as a DataSet object
DataSet.copy(self)	Make a deep copy of an DataSet object

17.2.1 nctoolkit.open_data

nctoolkit.**open_data**(*x=[]*, *checks=False*, ***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?
- ****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")
```

17.2.2 nctoolkit.open_url

`nctoolkit.open_url`(*x=None*, *ftp_details=None*, *wait=None*, *file_stop=None*)

Read netCDF data from a url as a DataSet object

Parameters

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

Returns **open_url**

Return type nctoolkit.DataSet

Examples

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

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

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

17.2.3 nctoolkit.open_thredds

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

Read thredds data as a DataSet object

Parameters

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

Returns **open_thredds**

Return type nctoolkit.DataSet

Examples

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

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

17.2.4 nctoolkit.DataSet.copy

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

Make a deep copy of an DataSet object

17.3 Merging or analyzing multiple datasets

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

17.3.1 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"].

17.3.2 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

17.3.3 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

17.4 Adding file(s) to a dataset

append

17.4.1 nctoolkit.append

Functions

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

17.5 Accessing attributes

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

17.5.1 nctoolkit.DataSet.variables

property DataSet.**variables**

List variables contained in a dataset

17.5.2 nctoolkit.DataSet.years

property DataSet.**years**

List years contained in a dataset

17.5.3 nctoolkit.DataSet.months

property DataSet.**months**

List months contained in a dataset

17.5.4 nctoolkit.DataSet.times

property DataSet.**times**

List times contained in a dataset

17.5.5 nctoolkit.DataSet.levels

property DataSet.**levels**

List levels contained in a dataset

17.5.6 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.

17.5.7 nctoolkit.DataSet.current

property DataSet.**current**

The current file or files in the DataSet object

17.5.8 nctoolkit.DataSet.history

property DataSet.**history**

The history of operations on the DataSet

17.5.9 nctoolkit.DataSet.start

property DataSet.**start**

The starting file or files of the DataSet object

17.6 Plotting

`DataSet.plot`(self[, vars])	

17.6.1 nctoolkit.DataSet.plot

`DataSet.plot`(*self*, *vars=None*)

17.7 Variable modification

`DataSet.assign`(self[, drop])	Create new variables Existing columns that are re-assigned will be overwritten. :param drop: Set to True if you want existing variables to be removed once the new ones have been created. Defaults to False.
`DataSet.rename`(self, newnames)	Rename variables in a dataset
`DataSet.set_missing`(self[, value])	Set the missing value for a single number or a range
`DataSet.sum_all`(self[, drop])	Calculate the sum of all variables for each time step

17.7.1 nctoolkit.DataSet.assign

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

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

Parameters **`**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.

17.7.2 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"})
```

17.7.3 nctoolkit.DataSet.set_missing

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

Set the missing value for a single number or a range

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.

17.7.4 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?

17.8 netCDF file attribute modification

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

17.8.1 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

17.8.2 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.

17.9 Vertical/level methods

`DataSet.surface`(self)	Extract the top/surface level from a dataset This extracts the first vertical level from each file in a dataset.
`DataSet.bottom`(self)	Extract the bottom level from a dataset This extracts the bottom level from each netCDF file.
`DataSet.vertical_interp`(self[, levels])	Verticaly interpolate a dataset based on given vertical levels This is calculated for each time step and grid cell
`DataSet.vertical_mean`(self)	Calculate the depth-averaged mean for each variable This is calculated for each time step and grid cell
`DataSet.vertical_min`(self)	Calculate the vertical minimum of variable values This is calculated for each time step and grid cell
`DataSet.vertical_max`(self)	Calculate the vertical maximum of variable values This is calculated for each time step and grid cell
`DataSet.vertical_range`(self)	Calculate the vertical range of variable values This is calculated for each time step and grid cell
`DataSet.vertical_sum`(self)	Calculate the vertical sum of variable values This is calculated for each time step and grid cell
`DataSet.vertical_cumsum`(self)	Calculate the vertical sum of variable values This is calculated for each time step and grid cell
`DataSet.invert_levels`(self)	Invert the levels of 3D variables This is calculated for each time step and grid cell
`DataSet.bottom_mask`(self)	Create a mask identifying the deepest cell without missing values.

17.9.1 nctoolkit.DataSet.surface

`DataSet.`**`surface`**`(`*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.surface()
```

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

17.9.2 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.

17.9.3 nctoolkit.DataSet.vertical_interp

`DataSet.`**`vertical_interp`**`(`*self*, *levels=None*`)`

Verticaly interpolate a dataset based on given vertical levels This is calculated for each time step and grid cell

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.

Examples

If you wanted to vertically interpolate a dataset to 5 and 10 metres, you would do the following:

```
>>> ds.vertical_interp([5,10])
```

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.

17.9.4 nctoolkit.DataSet.vertical_mean

DataSet.**vertical_mean**(*self*)

Calculate the depth-averaged mean for each variable This is calculated for each time step and grid cell

Examples

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

```
>>> ds.vertical_mean()
```

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.

17.9.5 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()
```

17.9.6 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()
```

17.9.7 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()
```

17.9.8 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()
```

17.9.9 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_sum()
```

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

17.9.10 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()
```

17.9.11 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

17.10 Rolling methods

DataSet.rolling_mean(self[, window])	Calculate a rolling mean based on a window
DataSet.rolling_min(self[, window])	Calculate a rolling minimum based on a window
DataSet.rolling_max(self[, window])	Calculate a rolling maximum based on a window
DataSet.rolling_sum(self[, window])	Calculate a rolling sum based on a window
DataSet.rolling_range(self[, window])	Calculate a rolling range based on a window

17.10.1 nctoolkit.DataSet.rolling_mean

DataSet.**rolling_mean**(*self*, *window=None*)

Calculate a rolling mean based on a window

Parameters = **int** (*window*) – The size of the window for the calculation of the rolling mean

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)
```

17.10.2 nctoolkit.DataSet.rolling_min

DataSet.**rolling_min**(*self*, *window=None*)

Calculate a rolling minimum based on a window

Parameters = **int** (*window*) – The size of the window for the calculation of the rolling minimum

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)
```

17.10.3 nctoolkit.DataSet.rolling_max

DataSet.**rolling_max**(*self*, *window=None*)

Calculate a rolling maximum based on a window

Parameters = **int** (*window*) – The size of the window for the calculation of the rolling maximum

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)
```

17.10.4 nctoolkit.DataSet.rolling_sum

DataSet.**rolling_sum**(*self*, *window=None*)

Calculate a rolling sum based on a window

Parameters = **int** (*window*) – The size of the window for the calculation of the rolling sum

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)
```

17.10.5 nctoolkit.DataSet.rolling_range

DataSet.**rolling_range**(*self*, *window=None*)

Calculate a rolling range based on a window

Parameters = **int** (*window*) – The size of the window for the calculation of the rolling range

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)
```

17.11 Evaluation setting

DataSet.run(self)	Run all stored commands in a dataset

17.11.1 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()
```

17.12 Cleaning functions

—

17.13 Ensemble creation

create_ensemble([path, recursive])	Generate an ensemble

17.13.1 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)
```

17.14 Arithemetic methods

`DataSet.assign`(self[, drop])	Create new variables Existing columns that are re-assigned will be overwritten. :param drop: Set to True if you want existing variables to be removed once the new ones have been created. Defaults to False.
`DataSet.add`(self[, x, var])	Add to a dataset This will add a constant, another dataset or a netCDF file to the dataset. :param x: 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. :type x: int, float, DataSet or netCDF file :param var: A variable in the x to use for the operation :type var: str.
`DataSet.subtract`(self[, x, var])	Subtract from a dataset This will subtract a constant, another dataset or a netCDF file from the dataset. :param x: 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. :type x: int, float, DataSet or netCDF file :param var: A variable in the x to use for the operation :type var: str.
`DataSet.multiply`(self[, x, var])	Multiply a dataset This will multiply a dataset by a constant, another dataset or a netCDF file. :param x: 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. :type x: int, float, DataSet or netCDF file :param var: A variable in the x to multiply the dataset by :type var: str.
`DataSet.divide`(self[, x, var])	Divide the data This will divide the dataset by a constant, another dataset or a netCDF file. :param x: 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. :type x: int, float, DataSet or netCDF file :param var: A variable in the x to use for the operation :type var: str.

17.14.1 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. :param x: 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.

Parameters **`var`** (*`str`*) – A variable in the x to use for the operation

Examples

If you wanted to add 10 to all variables in a dataset, you would do the following:

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

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

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

Grids in the datasets must match. Addition will occur in matching timesteps in ds1 and ds2. If there is only 1 timestep in ds2, then the data from that timestep will be added to the data in all ds1 time steps.

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

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

17.14.2 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. :param x: 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.

Parameters **`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 substract 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")
```

17.14.3 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. :param x: 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.

Parameters **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")
```

17.14.4 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. :param x: 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.

Parameters **var** (*str*) – A variable in the x to use for the operation

Examples

If you wanted to dividie 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")
```

17.15 Ensemble statistics

DataSet.ensemble_mean(self[, nco, ignore_time])	Calculate an ensemble mean
DataSet.ensemble_min(self[, nco, ignore_time])	Calculate an ensemble min
DataSet.ensemble_max(self[, nco, ignore_time])	Calculate an ensemble maximum
DataSet.ensemble_percentile(self[, p])	Calculate an ensemble percentile This will calculate the percentles for each time step in the files.
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.
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.

17.15.1 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.

17.15.2 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.

17.15.3 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.

17.15.4 nctoolkit.DataSet.ensemble_percentile

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

Calculate an ensemble percentile This will calculate the percentles 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<=p<=100.

17.15.5 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.

17.15.6 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.

17.16 Subsetting operations

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

17.16.1 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])
```

17.16.2 nctoolkit.DataSet.select

`DataSet.`**`select`**`(`*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

seasons [str] Seasons to select. One of “DJF”, “MAM”, “JJA”, “SON”.

months [list, range or int] Month(s) to select.

years [list,range or int] Years(s) to select. These should be integers

timesteps [list or int] time step(s) to select. For example, if you wanted the first time step set times=0.

Examples

If you want to select a single variable do the following:

```
>>> ds.select(variable = "var")
```

If you want to select a list of variables, do this:

```
>>> ds.select(variable = ["var1", "var2"])
```

If you want to select data for January, do the following:

```
>>> ds.select(month = 1)
```

If you want to select a range of months, do the following:

```
>>> ds.select(months = range(1, 7))
```

If you want to select a range of years, for example the 2010s, do the following:

```
>>> ds.select(years = range(2010, 2020))
```

If you want to select the first two timesteps in a dataset, do the following:

```
>>> ds.select(timesteps = [0,1])
```

17.16.3 nctoolkit.DataSet.drop

`DataSet.`**`drop`**(*self*, *vars=None*)

Remove variables This will remove stated variables from files in the dataset.

Parameters **`vars`** (*`str or list`*) – Variable or variables to be removed from the dataset. Variables that are listed but not in the dataset will be ignored

Examples

If you wanted to remove a single variable 'var1' from a dataset data, you would do the following:

```
>>> ds.drop('var')
```

If you wanted to remove a list of variables, you would do the following:

```
>>> ds.drop(['var1', 'var2', 'var2'])
```

17.17 Time-based methods

`DataSet.set_date`(self[, year, month, day, . . .])	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.
`DataSet.shift`(self, **kwargs)	Shift method.

17.17.1 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.

17.17.2 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)
```

17.18 Interpolation and resampling methods

DataSet.regrid(self[, grid, method, recycle])	Regrid a dataset to a target grid
DataSet.to_latlon(self[, lon, lat, res, . . .])	Regrid a dataset to a regular latlon grid
DataSet.resample_grid(self[, factor])	Resample the horizontal grid of a dataset
DataSet.time_interp(self[, start, end, . . .])	Temporally interpolate variables based on date range and time resolution
DataSet.timestep_interp(self[, steps])	Temporally interpolate a dataset to given number of time steps between existing time steps

17.18.1 nctoolkit.DataSet.regrid

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

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"

17.18.2 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

17.18.3 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)
```

17.18.4 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.

17.18.5 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.

17.19 Masking methods

DataSet.mask_box(self[, lon, lat])	Mask a lon/lat box

17.19.1 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]

17.20 Statistical methods

`DataSet.tmean`(self[, over])	Calculate the temporal mean of all variables
`DataSet.tmin`(self[, over])	Calculate the temporal minimum of all variables
`DataSet.tmedian`(self[, over])	Calculate the temporal median of all variables :param over: Time periods to average over.
`DataSet.tpercentile`(self[, p, over])	Calculate the temporal percentile of all variables
`DataSet.tmax`(self[, over])	Calculate the temporal maximum of all variables
`DataSet.tsum`(self[, over])	Calculate the temporal sum of all variables
`DataSet.trange`(self[, over])	Calculate the temporal range of all variables
`DataSet.tvariance`(self[, over])	Calculate the temporal variance of all variables
`DataSet.tstdev`(self[, over])	Calculate the temporal standard deviation of all variables
`DataSet.tcumsum`(self)	Calculate the temporal cumulative sum of all variables
`DataSet.cor_space`(self[, var1, var2])	Calculate the correlation correct between two variables in space This is calculated for each time step.
`DataSet.cor_time`(self[, var1, var2])	Calculate the correlation correct in time between two variables The correlation is calculated for each grid cell, ignoring missing values.
`DataSet.spatial_mean`(self)	Calculate the area weighted spatial mean for all variables This is performed for each time step.
`DataSet.spatial_min`(self)	Calculate the spatial minimum for all variables This is performed for each time step.
`DataSet.spatial_max`(self)	Calculate the spatial maximum for all variables This is performed for each time step.
`DataSet.spatial_percentile`(self[, p])	Calculate the spatial sum for all variables This is performed for each time step.
`DataSet.spatial_range`(self)	Calculate the spatial range for all variables This is performed for each time step.
`DataSet.spatial_sum`(self[, by_area])	Calculate the spatial sum for all variables This is performed for each time step.
`DataSet.centre`(self[, by, by_area])	Calculate the latitudinal or longitudinal centre for each year/month combination in files. This applies to each file in an ensemble. by : str Set to 'latitude' if you want the latitiduinal centre calculated. 'longitude' for longitudinal. by_area : bool If the variable is a value/m2 type variable, set to True, otherwise set to False.
`DataSet.zonal_mean`(self)	Calculate the zonal mean for each year/month combination in files.

continues on next page

Table 21 – continued from previous page

DataSet.zonal_min(self)	Calculate the zonal minimum for each year/month combination in files.
DataSet.zonal_max(self)	Calculate the zonal maximum for each year/month combination in files.
DataSet.zonal_range(self)	Calculate the zonal range for each year/month combination in files.
DataSet.meridonial_mean(self)	Calculate the meridonial mean for each year/month combination in files.
DataSet.meridonial_min(self)	Calculate the meridonial minimum for each year/month combination in files.
DataSet.meridonial_max(self)	Calculate the meridonial maximum for each year/month combination in files.
DataSet.meridonial_range(self)	Calculate the meridonial range for each year/month combination in files.

17.20.1 nctoolkit.DataSet.tmean

DataSet.**tmean**(*self*, *over='time'*)

Calculate the temporal mean of all variables

Parameters **over** (*str or list*) – Time periods to average over. Options are 'year', 'month', 'day'.

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")
```

17.20.2 nctoolkit.DataSet.tmin

`DataSet.`**`tmin`**`(`*self*, *over='time'*`)`

Calculate the temporal minimum of all variables

Parameters **`over`** (*`str or list`*) – Time periods to average over. Options are 'year', 'month', 'day'.

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")
```

17.20.3 nctoolkit.DataSet.tmedian

`DataSet.`**`tmedian`**`(`*self*, *over='time'*`)`

Calculate the temporal median of all variables :param over: Time periods to average over. Options are 'year', 'month', 'day'. :type over: str or list

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")
```

17.20.4 nctoolkit.DataSet.tpercentile

`DataSet.`**`tpercentile`**(*self*, *p=None*, *over='time'*)

Calculate the temporal percentile of all variables

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

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)
```

17.20.5 nctoolkit.DataSet.tmax

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

Calculate the temporal maximum of all variables

Parameters **`over`** (*`str or list`*) – Time periods to average over. Options are 'year', 'month', 'day'.

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")
```

17.20.6 nctoolkit.DataSet.tsum

`DataSet.`**`tsum`**(*self*, *over='time'*)

Calculate the temporal sum of all variables

17.20.7 nctoolkit.DataSet.trange

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

Calculate the temporal range of all variables

Parameters **`over`** (*`str or list`*) – Time periods to average over. Options are 'year', 'month', 'day'.

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")
```

17.20.8 nctoolkit.DataSet.tvariance

`DataSet.`**`tvariance`**(*self*, *over='time'*)

Calculate the temporal variance of all variables

Parameters **`over`** (*`str or list`*) – Time periods to average over. Options are 'year', 'month', 'day'.

Examples

If you want to calculate variance over all time steps. Do the following:

```
>>> ds.tvariance()
```

If you want to calculate the variance for each year in a dataset, do this:

```
>>> ds.tvariance("year")
```

If you want to calculate the variance for each month in a dataset, do this:

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

If you want to calculate the variance for each month in each year in a dataset, do this:

```
>>> ds.tvariance(["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.tvariance( "month")
```

A daily climatological variance would be the following:

```
>>> ds.tvariance( "day")
```

17.20.9 nctoolkit.DataSet.tstdev

`DataSet.`**`tstdev`**(*self*, *over='time'*)

Calculate the temporal standard deviation of all variables

Parameters **`over`** (*`str or list`*) – Time periods to average over. Options are 'year', 'month', 'day'.

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")
```

17.20.10 nctoolkit.DataSet.tcumsum

`DataSet.`**`tcumsum`**`(`*self*`)`

Calculate the temporal cumulative sum of all variables

Examples

If you want to calculate the cumulative sum for all variables over all timesteps, do this:

```
>>> ds.tcumsum()
```

17.20.11 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 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.

17.20.12 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.

17.20.13 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.

17.20.14 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()
```

17.20.15 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()
```

17.20.16 nctoolkit.DataSet.spatial_percentile

`DataSet.`**`spatial_percentile`**`(`*self, p=None*`)`

Calculate the spatial sum for all variables This is performed for each time step. :param p: Percentile to calculate. 0<=p<=100. :type p: int or float

Examples

If you want to calculate the median of each variable across space for a dataset, just do the following:

```
>>> ds.spatial_percentile(50)
```

17.20.17 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()
```

17.20.18 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.

17.20.19 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. by : str

Set to 'latitude' if you want the latitiduinal centre calculated. 'longitude' for longitudinal.

by_area [bool] If the variable is a value/m2 type variable, set to True, otherwise set to False.

17.20.20 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()
```

17.20.21 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()
```

17.20.22 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()
```

17.20.23 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()
```

17.20.24 nctoolkit.DataSet.meridonial_mean

DataSet.**meridonial_mean**(*self*)

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

Examples

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

```
>>> ds.meridonial_mean()
```

17.20.25 nctoolkit.DataSet.meridonial_min

DataSet.**meridonial_min**(*self*)

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

Examples

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

```
>>> ds.meridonial_min()
```

17.20.26 nctoolkit.DataSet.meridonial_max

DataSet.**meridonial_max**(*self*)

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

Examples

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

```
>>> ds.meridonial_max()
```

17.20.27 nctoolkit.DataSet.meridonial_range

DataSet.**meridonial_range**(*self*)

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

Examples

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

```
>>> ds.meridonial_max()
```

17.21 Merging methods

DataSet.merge(self[, match])	Merge a multi-file ensemble into a single file Merging will occur based on the time steps in the first file.
DataSet.merge_time(self)	Time-based merging of a multi-file ensemble into a single file This method is ideal if you have the same data split over multiple files covering different data sets.

17.21.1 nctoolkit.DataSet.merge

DataSet.**merge**(*self*, *match=['year', 'month', 'day']*)

Merge a multi-file ensemble into a single file Merging will occur based on the time steps in the first file. This will only be effective if you want to merge files with the same times, but with different variables.

Parameters **match** (*list, str*) – 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.

17.21.2 nctoolkit.DataSet.merge_time

DataSet.**merge_time**(*self*)

Time-based merging of a multi-file ensemble into a single file This method is ideal if you have the same data split over multiple files covering different data sets.

17.22 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.

17.22.1 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'. 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.

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")
```

17.23 Output and formatting methods

DataSet.to_nc(self, out[, zip, overwrite])	Save a dataset to a named file This will only work with single file datasets.
DataSet.to_xarray(self[, decode_times, . . .])	Open a dataset as an xarray object
DataSet.to_dataframe(self[, decode_times, . . .])	Open a dataset as a pandas data frame
DataSet.zip(self)	Zip the dataset This will compress the files within the dataset.
DataSet.format(self[, ext])	Zip the dataset This will compress the files within the dataset. This works lazily. :param ext: New format. Must be one of "nc", "nc1", "nc2", "nc4" and "nc5" . netCDF = nc1 netCDF version 2 (64-bit offset) = nc2/nc netCDF4 (HDF5) = nc4 netCDF4-classi = nc4c netCDF version 5 (64-bit data) = nc5 :type ext: str.

17.23.1 nctoolkit.DataSet.to_nc

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

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.

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)
```

17.23.2 nctoolkit.DataSet.to_xarray

`DataSet.`**`to_xarray`**(*self*, *decode_times=True*, *cdo_times=False*)

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.
- **`cdo_times`** (*`boolean`*) – Set to True if you do not want CDO to decode the times

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)
```

17.23.3 nctoolkit.DataSet.to_dataframe

`DataSet.`**`to_dataframe`**(*self*, *decode_times=True*, *cdo_times=False*)

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.
- **`cdo_times`** (*`boolean`*) – Set to True if you do not want CDO to decode the times

Returns to_dataframe

Return type pandas.DataFrame

17.23.4 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.

17.23.5 nctoolkit.DataSet.format

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

Zip the dataset This will compress the files within the dataset. This works lazily. :param ext: New format. Must be one of "nc", "nc1", "nc2", "nc4" and "nc5" .

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

17.24 Miscellaneous methods

DataSet.cell_area(self[, join])	Calculate the area of grid cells.
DataSet.first_above(self[, x])	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. :param x: 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. :type x: int, float, DataSet or netCDF file.
DataSet.first_below(self[, x])	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. :param x: 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. :type x: int, float, DataSet or netCDF file.
DataSet.last_above(self[, x])	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. :param x: 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. :type x: int, float, DataSet or netCDF file.
DataSet.last_above(self[, x])	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. :param x: 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. :type x: int, float, DataSet or netCDF file.

continues on next page

Table 25 – continued from previous page

DataSet.cdo_command(self[, command])	Apply a cdo command
DataSet.nco_command(self[, command, ensemble])	Apply an nco command
DataSet.compare(self[, expression])	Compare all variables to a constant
DataSet.gt(self, x)	Method to calculate if variable in dataset is greater than that in another file or dataset This currently only works with single file datasets
DataSet.lt(self, x)	Method to calculate if variable in dataset is less than that in another file or dataset This currently only works with single file datasets
DataSet.reduce_dims(self)	Reduce dimensions of data This will remove any dimensions with only one value.
DataSet.reduce_grid(self[, mask])	Reduce the dataset to non-zero locations in a mask :param mask: single variable dataset or path to .nc file. The mask must have an identical grid to the dataset. :type mask: str or dataset.

17.24.1 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)
```

17.24.2 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. :param x: 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)
```

17.24.3 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. :param x: 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)
```

17.24.4 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. :param x: 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)
```

17.24.5 nctoolkit.DataSet.cdo_command

`DataSet.`**`cdo_command`**(*self*, *command=None*)

Apply a cdo command

Parameters **`command`** (*`string`*) – cdo command to call. This command must be such that "cdo {command} infile outfile" will run.

17.24.6 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.

17.24.7 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 calculcated for each time step.

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

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

17.24.8 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

17.24.9 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

17.24.10 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.

17.24.11 nctoolkit.DataSet.reduce_grid

`DataSet.reduce_grid(self, mask=None)`

Reduce the dataset to non-zero locations in a mask :param mask: single variable dataset or path to .nc file.

The mask must have an identical grid to the dataset.

17.25 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.

17.25.1 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.

CHAPTER

EIGHTEEN

PACKAGE INFO

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

18.1 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.

18.2 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.

18.3 Contributions welcome

The package is new, with new features being added each month. There remain a large number of features that could be added, especially for dealing with atmospheric data. If packages users are interested in contributing or suggesting new features they are welcome to raise and issue at the package's GitHub page or contact me.

PYTHON MODULE INDEX

INDEX