nctoolkit

Robert Wilson

Feb 02, 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

FIXING PLOTTING PROBLEM DUE TO XARRAY BUG

There is currently a bug in xarray caused by the update of pandas to version 1.1. As a result some plots will fail in nctoolkit. To fix this ensure pandas version 1.0.5 is installed. Do this after installing nctoolkit. This can be done as follows:

```
$ conda install -c conda-forge pandas=1.0.5
```

or:

```
$ pip install pandas==1.0.5
```

CHAPTER

TWO

DOCUMENTATION

Quick overview

- *Installation*
- *Introduction tutorial*
- *Ensemble methods*
- lazy_methods
- *News*

2.1 Installation

2.1.1 Python dependencies

- Python (3.6 or later)
- numpy (1.14 or later)
- pandas (0.24 or later)
- xarray (0.14 or later)
- hvplot (0.5 or later)
- NetCDF4 (1.53 or later)
- panel (0.9.1 or later)

2.1.2 How to install nctoolkit

The easiest way to install the package is using conda, which will install nctoolkit and all system dependencies:

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

nctoolkit is available from the Python Packaging Index. To install nctoolkit using pip:

```
$ pip install nctoolkit
```

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

To install the development version from GitHub:

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

2.1.3 Fixing plotting problem due to xarray bug

There is currently a bug in xarray caused by the update of pandas to version 1.1. As a result some plots will fail in nctoolkit. To fix this ensure pandas version 1.0.5 is installed. Do this after installing nctoolkit. This can be done as follows:

```
$ conda install -c conda-forge pandas=1.0.5
```

or:

```
$ pip install pandas==1.0.5
```

2.1.4 System dependencies

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

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

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

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

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

2.2 Introduction tutorial

nctoolkit is designed for the efficient analysis and manipulation of netCDF files. This tutorial provides an overview of how to work with individual files.

2.2.1 Opening netcdf data

This tutorial will illustrate the basic usage using a dataset of average global sea surface temperature from NOAA, which is available here.

nctoolkit should be imported using the nc shorthand:

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

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

Reading in a dataset is straightforward:

[2]:
```
ff =  "sst.mon.ltm.1981-2010.nc"
sst = nc.open_data(ff)
```

We might want to know some basic information about the file. This can be done easily. Listing the available variables can be found quickly:

The current state of the dataset can be found as follows:

[3]:
```
sst.variables
```

[3]:
```
['sst', 'valid_yr_count']
```

The months available can be found using:

[4]:
```
sst.months
```

[4]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

We have 12 months available. In this case it is the monthly average temperature from 1981-2010.

2.2.2 Modifying datasets

Each time nctoolkit executes a command that modifies a dataset, it will generate a new NetCDF file, which becomes the `current` file in the dataset. Before any modification this is as follows:

[5]:
```
sst.current
```

[5]:
```
'sst.mon.ltm.1981-2010.nc'
```

We have seen that there are two variables in the dataset. But we only really care about `sst`. So let's select that variable:

[6]:
```
sst.select(variables = "sst")
```

We can now see that there is only one variable in the sst dataset

[7]:
```
sst.variables
```

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

We can also that a temporary file has been created with only this variable in it

[8]:
```
sst.current
```

[8]:
```
'/tmp/nctoolkitibehjxqnnctoolkittmpj1d6le2g.nc'
```

We have data for 12 months. But what we might really want is an average of those values. This can be quickly calculated:

[9]:
```
sst.tmean()
```

Once again a new temporary file has been generated.

[10]:
```
sst.current
```

[10]:
```
'/tmp/nctoolkitibehjxqnnctoolkittmpmbbax0mt.nc'
```

Do not worry about the temporary folder getting clogged up. nctoolkit cleans it up automatically.

Quick visualization of netCDF data is always a good thing. So nctoolkit provides an easy autoplot feature.

[11]:
```
sst.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

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

```
Unable to decode time axis into full numpy.datetime64 objects, continuing using␣
↪cftime.datetime objects instead, reason: dates out of range
Unable to decode time axis into full numpy.datetime64 objects, continuing using␣
↪cftime.datetime objects instead, reason: dates out of range
```

[11]:
```
:DynamicMap   []
   :Overlay
      .Image.I     :Image   [lon,lat]   (sst)
      .Coastline.I :Feature   [Longitude,Latitude]
```

What we have seen so far is not computionally efficient. In the code below nctoolkit has generated temporary files twice:

[12]:
```
sst = nc.open_data(ff)
sst.select(variables = "sst")
sst.tmean()
```

We can see what went on behind the scenes by accessing `history`:

[13]:
```
sst.history
```

[13]:
```
['cdo -L -selname,sst sst.mon.ltm.1981-2010.nc /tmp/
↪nctoolkitibehjxqnnctoolkittmptqwymkom.nc',
 'cdo -L -timmean /tmp/nctoolkitibehjxqnnctoolkittmptqwymkom.nc /tmp/
↪nctoolkitibehjxqnnctoolkittmp0j30x01r.nc']
```

nctoolkit uses CDO. You do not understand how CDO works to use nctoolkit. But one nice feature of CDO is method chaining, which works like Python's. To get it working you just need to set evaluation to lazy in nctoolkit. This means nothing is evaluated until you force it to or it has to be.

[14]:
```
nc.options(lazy = True)
```

Now, let's run the code again:

[15]:
```
sst = nc.open_data(ff)
sst.select(variables = "sst")
sst.tmean()
sst.plot()
```

```
Unable to decode time axis into full numpy.datetime64 objects, continuing using␣
↪cftime.datetime objects instead, reason: dates out of range
Unable to decode time axis into full numpy.datetime64 objects, continuing using␣
↪cftime.datetime objects instead, reason: dates out of range
```

[15]:
```
:DynamicMap   []
   :Overlay
      .Image.I     :Image    [lon,lat]    (sst)
      .Coastline.I :Feature    [Longitude,Latitude]
```

When we look at `history`, we now see that only one temporary file was generated:

[16]:
```
sst.history
```

[16]:
```
['cdo -L -timmean -selname,sst sst.mon.ltm.1981-2010.nc /tmp/
↪nctoolkitibehjxqnnctoolkittmpm0u39jql.nc']
```

In the example, above the commands were only executed when plot was called. If we want to force commands to run we use `run`:

[17]:
```
sst = nc.open_data(ff)
sst.select_variables("sst")
sst.mean()
sst.run()
```

2.3 News

2.3.1 Release of v0.3.0

Version 0.3.0 will be 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.

User Guide

- *Datasets*

2.4 Datasets

nctoolkit works with what it calls datasets. Each dataset is made up of a single or multiple NetCDF files. Each time you apply a method to a dataset the NetCDF file or files within the dataset will be modified.

2.4.1 Opening datasets

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

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

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

2.4.2 Dataset attributes

We can find out key information about a dataset using its attributes. Here we will use a sea surface temperature file that is available via thredds.

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

```
data.variables
```

If we want to know the vertical levels available in the dataset, we use the following. This is particularly useful for oceanic data.

```
data.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.

```
data.current
```

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

```
data.times
```

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

```
data.months
```

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

```
data.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:

```
data.history
```

2.4.3 Lazy evaluation of datasets

nctoolkit works by performing operations and then saving the results as either a temporary file or in a file specified by the user. 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:

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

What is potentially wrong with? 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.

We force the lines to be evaluated using `run`:

```
data.history
```

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

If we working in a Jupyter notebook, we could instead use `plot` at the end of the chain:

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

This will force everything to be evaluated before plotting.

An alternative will be to write to a results file at the end of the chain:

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

This creates an ultra-efficient processing chain where we read the input file and write to the output file with no intermediate file writing.

2.4.4 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:

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

2.4.5 List-like behaviour of datasets

Datasets can be made up of multi-files. To make processing these files easier nctoolkit features a number of methods similar to lists.

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`:

2.5 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
data = nc.open_data("sst.mon.mean.nc")
data.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:

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

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

```
data.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.

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

2.5.1 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:

```
data.rolling_mean(7)
```

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

```
data.rolling_sum(7)
```

2.5.2 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:

```
data.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:

```
data.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.

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

Monthly anomalies are calculated in the same way:

```
data.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.

2.5.3 Calculating climatologies

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

```
data.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:

```
data.tmean("Seas")
```

Calculating a climatological monthly mean would require the following:

```
data.tmean("month")
```

and daily would be the following:

```
data.tmean("day")
```

2.5.4 Calculating climatologies

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

```
data.tmean("season")
```

2.5.5 Cumulative sums

We can calculate the cumulative sum as follows:

```
data.tcumsum()
```

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

2.6 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.

2.6.1 Selecting variables

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

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

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

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

2.6.2 Selecting years

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

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

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

```
data.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:

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

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

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

2.6.3 Selecting months

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

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

2.6.4 Selecting seasons

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

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

2.6.5 Selecting timesteps

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

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

2.6.6 Geographic subsetting

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

```
data.crop(lon = [-80, 90], lat = [50, 80])
```

2.7 Creating 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:

```
data.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:

```
data.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

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

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

2.7.1 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:

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

An example of using logs would be the following:

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

2.7.2 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:

```
data.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:

```
data.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:

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

or like this:

```
data.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:

```
data.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:

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

2.7.3 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)

2.8 Importing and exporting data

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

2.8.1 Opening single files and ensembles

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

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

2.8.2 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
data = nc.open_url(example_url)
```

2.8.3 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
data = nc.open_thredds(example_url)
```

2.8.4 Exporting datasets

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

2.8.5 Save as a NetCDF

The method `write_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 `write_nc`. An example of usage is as follows:

```
data = nc.open_data(infile)
data.tmean()
data.zip()
data.write_nc(outfile)
```

2.8.6 Convert to xarray Dataset

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

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

2.8.7 Convert to pandas dataframe

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

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

2.9 Ensemble methods

2.9.1 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.

2.9.2 Merging files with different times

TBC

2.9.3 Ensemble averaging

TBC

2.10 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.

2.10.1 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.

2.10.2 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):
    data = nc.open_data(ff)
    data.assign(tos = lambda x: x.sst + 273.15)
    data.tmean()
    data.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.

2.11 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])
```

(continues on next page)

(continued from previous page)

```
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")
```

2.11.1 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.

Reference and help

- *An A-Z guide to nctoolkit methods*
- *API Reference*
- *How to guide*
- *Package info*

2.12 An A-Z guide to nctoolkit methods

This guide will provide examples of how to use almost every method available in nctoolkit.

2.12.1 add

This method can add to a dataset. You can add a constant, another dataset or a NetCDF file. In the case of datasets or NetCDF files the grids etc. must be of the same structure as the original dataset.

For example, if we had a temperature dataset where temperature was in Celsius, we could convert it to Kelvin by adding 273.15.

```
data.add(273.15)
```

If we have two datasets, we add one to the other as follows:

```
data1 = nc.open_data(infile1)
data2 = nc.open_data(infile2)
data1.add(data2)
```

In the above example, all we are doing is adding infile2 to data2, so instead we could simply do this:

```
data1.add(infile2)
```

2.12.2 annual_anomaly

This method will calculate the annual anomaly for each variable (and in each grid cell) compared with a baseline. This is a standard anomaly calculation where first the mean value is calculated for the baseline period, and the difference between the values is calculated.

For example, if we wanted to calculate the anomalies in a dataset compared with a baseline period of 1900-1919 we would do the following:

```
data.annual_anomaly(baseline=[1900, 1919])
```

We may be more interested in the rolling anomaly, in particular when there is a lot of annual variation. In the above case, if you wanted a 20 year rolling mean anomaly, you would do the following:

```
data.annual_anomaly(baseline=[1900, 1919], window=20)
```

By default this method works out the absolute anomaly. However, in some cases the relative anomaly is more interesting. To calculate this we set the metric argument to “relative”:

```
data.annual_anomaly(baseline=[1900, 1919], metric = "relative")
```

2.12.3 annual_max

This method will calculate the maximum value in each available year and for each grid cell of dataset.

```
data.annual_max()
```

2.12.4 annual_mean

This method will calculate the maximum value in each available year and for each grid cell of dataset.

```
data.annual_mean()
```

2.12.5 annual_min

This method will calculate the minimum value in each available year and for each grid cell of dataset.

```
data.annual_min()
```

2.12.6 annual_range

This method will calculate the range of values in each available year and for each grid cell of dataset.

```
data.annual_range()
```

2.12.7 annual_sum

This method will calculate the sum of values in each available year and for each grid cell of dataset.

```
data.annual_sum()
```

2.12.8 append

This method will let you append individual or multiple files to your dataset. Usage is straightforward. Note that this will not perform any merging on the dataset.

```
data.append(newfile)
```

2.12.9 bottom

This method will extract the bottom vertical level from a dataset. This is useful for some oceanographic datasets, where the method can let you select the seabed. Note that this method will not work with all data types. For example, in ocean data with fixed depth levels, the bottom cell in the NetCDF data is not the actual seabed. See bottom_mask for these cases.

```
data.bottom()
```

2.12.10 bottom_mask

This method will identify the bottommost level in each grid with a non-NA value.

```
data.bottom_mask()
```

2.12.11 cdo_command

This method let's you run a cdo command. CDO commands are generally of the form "cdo {command} infile outfile". cdo_command therefore only requires the command portion of this. If we wanted to run the following CDO command

```
cdo -timmean -selmon,4 infile outfile
```

we would do the following:

```
data.cdo_command("-timmean -selmon,4")
```

2.12.12 cell_areas

This method either adds the areas of each grid cell to the dataset or converts the dataset to a new dataset showing only the grid cell areas. By default it adds the cell areas (in square metres) to the dataset.

```
data.cell_areas()
```

If we only want the cell areas we can set join to False:

```
data.cell_areas(join=False)
```

2.12.13 centre

This method calculates the longitudinal or latitudinal centre of a dataset. There is one argument, which should either be "latitude" or "longitude". If you want to calculate the latitudinal centre:

```
data.centre("longitude")
```

2.12.14 crop

This method will crop a region to a specified longitude and latitude box. For example, if we wanted to crop a dataset to the North Atlantic, we could do this:

```
data.crop(lon = [-80, 20], lat = [40, 70])
```

2.12.15 compare_all

This method let's us compare all variables in a dataset with a constant. If we wanted to identify the grid cells with values above 20, we could do the following:

```
data.compare_all(">20")
```

Similarly, if we wanted to identify grid cells with negative values we would do this:

```
data.compare_all("<0")
```

2.12.16 cor_space

This method calculates the correlation coefficients between two variables in space for each time step. So, if we wanted to work out the correlation between the variables var1 and var2, we would do this:

```
data.cor_space("var1", "var2")
```

2.12.17 cor_time

This method calculates the correlation coefficients between two variables in time for each grid cell. If we wanted to work out the correlation between two variables var1 and var2 we would do the following:

```
data.cor_time("var1", "var2")
```

2.12.18 cum_sum

This method will calculate the cumulative sum, over time, for all variables. Usage is simple:

```
data.cum_sum()
```

2.12.19 daily_max

This method will calculate the maximum value in each available day and for each grid cell of dataset.

```
data.daily_max()
```

2.12.20 daily_mean

This method will calculate the maximum value in each available day and for each grid cell of dataset.

```
data.daily_mean()
```

2.12.21 daily_min

This method will calculate the minimum value in each available day and for each grid cell of dataset.

```
data.daily_min()
```

2.12.22 daily_range

This method will calculate the range of values in each available day and for each grid cell of dataset.

```
data.daily_range()
```

2.12.23 daily_sum

This method will calculate the sum of values in each available day and for each grid cell of dataset.

```
data.daily_sum()
```

2.12.24 daily_max_climatology

This method will calculate the maximum value that is observed on each day of the year over time. So, for example, if you had 100 years of daily temperature data, it will calculate the maximum value ever observed on each day.

```
data.daily_max_climatology()
```

2.12.25 daily_mean_climatology

This method will calculate the mean value that is observed on each day of the year over time. So, for example, if you had 100 years of daily temperature data, it will calculate the mean value ever observed on each day.

```
data.daily_mean_climatology()
```

2.12.26 daily_min_climatology

This method will calculate the minimum value that is observed on each day of the year over time. So, for example, if you had 100 years of daily temperature data, it will calculate the minimum value ever observed on each day.

```
data.daily_min_climatology()
```

2.12.27 daily_range_climatology

This method will calculate the value range that is observed on each day of the year over time. So, for example, if you had 100 years of daily temperature data, it will calculate the difference between the maximum and minimum observed values each day.

:: data.daily_range_climatology()

2.12.28 divide

This method will divide a dataset by a constant, or the values in another dataset of NetCDF file. If we wanted to divide everything in a dataset by 2, we would do the following:

```
data.divide(2)
```

If we want to divide a dataset by another, we can do this easily. Note that the datasets must be comparable, i.e. they must have the same grid. The second dataset must have either the same number of variables or only one variable. In the latter case everything is divided by that variable. The same holds for vertical levels.

```
data1 = nc.open_data(infile1)
data2 = nc.open_data(infile2)
data1.divide(data2)
```

2.12.29 ensemble_max, ensemble_min, ensemble_range and ensemble_mean

These methods will calculate the ensemble statistic, when a dataset is made up of multiple files. Two methods are available. First, the statistic across all available time steps can be calculated. For this ignore_time must be set to False. For example:

```
data = nc.open_data(file_list)
data.ensemble_max(ignore_time = True)
```

The second method is to calculate the maximum value in each given time step. For example, if the ensemble was made up of 100 files where each file contains 12 months of data, ensemble_max will work out the maximum monthly value. By default ignore_time is False.

```
data = nc.open_data(file_list)
data.ensemble_max(ignore_time = False)
```

2.12.30 ensemble_percentile

This method works in the same way as ensemble_mean etc. above. However, it requires an additional term p, which is the percentile. For example, if we had to calculate the 75th ensemble percentile, we would do the following:

```
data = nc.open_data(file_list)
data = nc.ensemble_percentile(75)
```

2.12.31 format

This method will change the format of the files within a dataset. For example if you wanted to convert to NetCDF4:

```
data.format("nc4")
```

2.12.32 invert_levels

This method will invert the vertical levels of a dataset.

```
data.invert_levels()
```

2.12.33 mask_box

This method will set everything outside a specificied longitude/latitude box to NA. The code below illustrates how to mask the North Atlantic in the SST dataset.

```
data.mask_box(lon = [-80, 20], lat = [40, 70])
```

2.12.34 max

This method will calculate the maximum value of all variables in all grid cells. If we wanted to calculate the maximum observed monthly sea surface temperature in the SST dataset we would do the following:

```
data.max()
```

2.12.35 mean

This method will calculate the mean value (averaged across all time steps) of all variables in all grid cells. Usage is simple:

```
data.mean()
```

2.12.36 median

This method will calculate the median value (averaged across all time steps) of all variables in all grid cells. Usage is simple:

```
data.median()
```

2.12.37 merge and merge_time

nctoolkit offers two methods for merging the files within a multi-file dataset. These methods operate in a similar way to column based joining and row-based binding in dataframes.

The merge method is suitable for merging files that have different variables, but the same time steps. The merge_time method is suitable for merging files that have the same variables, but have different time steps.

Usage for merge_time is as simple as:

```
data = nc.open_data(file_list)
data.merge_time()
```

Merging NetCDF files with different variables is potentially risky, as it is possible you can merge files that have the same number of time steps but have different times. nctoolkit's merge method therefore offers some security against a major error when merging. It requires a match argument to be supplied. This ensures that the times in each file is comparable to the others. By default match = ["year", "month", "day"], i.e. it checks if the times in each file all have the same year, month and day. The match argument must be some subset of ["year", "month", "day"]. For example, if you wanted to only make sure the files had the same year, you would do the following:

```
data = nc.open_data(file_list)
data.merge(match = ["year", "month", "day"])
```

2.12.38 meridonial statistics

Calculate the following meridonial statistics: mean, min, max and range:

```
data.meridonial_mean()
data.meridonial_min()
data.meridonial_max()
data.meridonial_range()
```

2.12.39 min

This method will calculate the minimum value (across all time steps) of all variables in all grid cells. Usage is simple:

```
data.min()
```

2.12.40 monthly_anomaly

This method will calculate the monthly anomaly compared with the mean value for a baseline period. For example, if we wanted the monthly anomaly compared with the mean for 1990-1999 we would do the below.

```
data.monthly_anomaly(baseline = [1990, 1999])
```

2.12.41 monthly_max

This method will calculate the maximum value in the month of each year of a dataset. This is useful for daily time series. If you want to calculate the mean value in each month across all available years, use monthly_max_climatology. Usage is simple:

```
data.monthly_max()
```

2.12.42 monthly_max_climatology

This method will calculate, for each month, the maximum value of each variable over all time steps.

```
data.monthly_max_climatology()
```

2.12.43 monthly_mean

This method will calculate the mean value of each variable in each month of a dataset. Note that this is calculated for each year. See monthly_mean_climatology if you want to calculate a climatological monthly mean.

```
data.monthly_mean()
```

2.12.44 monthly_mean_climatology

This method will calculate, for each month, the maximum value of each variable over all time steps. Usage is simple:

```
data.monthly_mean_climatology()
```

2.12.45 monthly_min

This method will calculate the minimum value in the month of each year of a dataset. This is useful for daily time series. If you want to calculate the mean value in each month across all available years, use monthly_max_climatology. Usage is simple:

```
data.monthly_min()
```

2.12.46 monthly_min_climatology

This method will calculate, for each month, the minimum value of each variable over all time steps. Usage is simple:

```
data.monthly_min_climatology()
```

2.12.47 monthly_range

This method will calculate the value range in the month of each year of a dataset. This is useful for daily time series. If you want to calculate the value range in each month across all available years, use monthly_range_climatology. Usage is simple:

```
data.monthly_range()
```

2.12.48 monthly_range_climatology

This method will calculate, for each month, the value range of each variable over all time steps. Usage is simple:

```
data.monthly_range_climatology()
```

2.12.49 multiply

This method will multiply a dataset by a constant, another dataset or a NetCDF file. If multiplied by a dataset or NetCDF file, the dataset must have the same grid and can only have one variable.

If you want to multiply a dataset by 2, you can do the following:

```
data.multiply(2)
```

If you wanted to multiply a dataset data1 by another, data2, you can do the following:

```
data1 = nc.open_data(infile1)
data2 = nc.open_data(infile2)
data1.multiply(data2)
```

2.12.50 mutate

This method can be used to generate new variables using arithmetic expressions. New variables are added to the dataset. The method requires a dictionary, where the key-value pairs are the new variables and expression required to generate it.

For example, if had a temperature dataset, with temperature in Celsius, we might want to convert that to Kelvin. We can do this easily:

```
data.mutate({"temperature_k":"temperature+273.15"})
```

2.12.51 percentile

This method will calculate a given percentile for each variable and grid cell. This will calculate the percentile using all available timesteps.

We can calculate the 75th percentile of sea surface temperature as follows:

```
data.percentile(75)
```

2.12.52 phenology

A number of phenological indices can be calculated. These are based on the plankton metrics listed by Ji et al. 2010. These methods require datasets or the files within a dataset to only be made up of individual years, and ideally every day of year is available. At present this method can only calculate the phenology metric for a single variable.

The available metrics are: peak - the time of year when the maximum value of a variable occurs. middle - the time of year when 50% of the annual cumulative sum of a variable is first exceeded start - the time of year when a lower threshold (which must be defined) of the annual cumulative sum of a variable is first exceeded end - the time of year when an upper threshold (which must be defined) of the annual cumulative sum of a variable is first exceeded

For example, if you wanted to calculate timing of the peak, you set metric to "peak", and define the variable to be analyzed:

```
data.phenology(metric = "peak", var = "var_chosen")
```

2.12.53 plot

This method will plot the contents of a dataset. It will either show a map or a time series, depending on the data type. While it should work on at least 90% of NetCDF data, there are some data types that remain incompatible, but will be added to nctoolkit over time. Usage is simple:

```
data.plot()
```

2.12.54 range

This method calculates the range for all variables in each grid cell across all steps.

We can calculate the range of sea surface temperatures in the SST dataset as follows:

```
data.range()
```

2.12.55 regrid

This method will remap a dataset to a new grid. This grid must be either a pandas data frame, a NetCDF file or a single file nctoolkit dataset.

For example, if we wanted to regrid a dataset to a single location, we could do the following:

```
import pandas as pd
data = nc.open_data(infile)
grid = pd.DataFrame({"lon":[-20], "lat":[50]})
data.regrid(grid, method = "nn")
```

If we wanted to regrid one dataset, dataset1, to the grid of another, dataset2, using bilinear interpolation, we would do the following:

```
data1 = nc.open_data(infile1)
data2 = nc.open_data(infile2)
data1.regrid(data2, method = "bil")
```

2.12.56 remove_variables

This method will remove variables from a dataset. Usage is simple, with the method only requiring either a str of a single variable or a list of variables to remove:

```
data.remove_variables(vars)
```

2.12.57 rename

This method allows you to rename variables. It requires a dictionary, with key-value pairs representing the old variable names and new variables. For example, if we wanted to rename a variable old to new, we would do the following:

```
data.rename({"old":"new"})
```

2.12.58 resample_grid

This method let's you resample the horizontal grid. It takes one argument. If you wanted to only take every other grid cell, you would do the following:

```
data.resample_grid(2)
```

2.12.59 rolling_max

This method will calculate the rolling maximum over a specifiied window. For example, if you needed to calculate the rolling maximum with a window of 10, you would do the following:

```
data.rolling_max(window = 10)
```

2.12.60 rolling_mean

This method will calculate the rolling mean over a specifiied window. For example, if you needed to calculate the rolling mean with a window of 10, you would do the following:

```
data.rolling_mean(window = 10)
```

2.12.61 rolling_min

This method will calculate the rolling minimum over a specifiied window. For example, if you needed to calculate the rolling minimum with a window of 10, you would do the following:

```
data.rolling_min(window = 10)
```

2.12.62 rolling_range

This method will calculate the rolling range over a specifiied window. For example, if you needed to calculate the rolling range with a window of 10, you would do the following:

```
data.rolling_range(window = 10)
```

2.12.63 rolling_sum

This method will calculate the rolling sum over a specifiied window. For example, if you needed to calculate the rolling sum with a window of 10, you would do the following:

```
data.rolling_sum(window = 10)
```

2.12.64 run

This method will evaluate all of a dataset's unevaluated commands. Evaluation should be set to lazy. Usage is simple:

```
nc.options(lazy = True)
data = nc.open_data(infile)
#.... apply some methods to the dataset
data.run()
```

2.12.65 seasonal_max

This method will calculate the maximum value observed in each season. Note this is worked out for the seasons of each year. See seasonal_max_climatology for climatological seasonal maximums.

```
data.seasonal_max()
```

2.12.66 seasonal_max_climatology

This method calculates the maximum value observed in each season across all years. Usage is simple:

```
data.seasonal_max_climatology()
```

2.12.67 seasonal_mean

This method will calculate the mean value observed in each season. Note this is worked out for the seasons of each year. See seasonal_mean_climatology for climatological seasonal means.

```
data.seasonal_mean()
```

2.12.68 seasonal_mean_climatology

This method calculates the mean value observed in each season across all years. Usage is simple:

```
data.seasonal_mean_climatology()
```

2.12.69 seasonal_min

This method will calculate the minimum value observed in each season. Note this is worked out for the seasons of each year. See seasonal_min_climatology for climatological seasonal minimums.

```
data.seasonal_min()
```

2.12.70 seasonal_min_climatology

This method calculates the minimum value observed in each season across all years. Usage is simple:

```
data.seasonal_min_climatology()
```

2.12.71 seasonal_range

This method will calculate the value range observed in each season. Note this is worked out for the seasons of each year. See seasonal_range_climatology for climatological seasonal ranges.

```
data.seasonal_range()
```

2.12.72 seasonal_range_climatology

This method calculates the value range observed in each season across all years. Usage is simple:

```
data.seasonal_range_climatology()
```

2.12.73 select

A method to subset a dataset based on multiple criteria. This acts as a wrapper for *select_variables*, *select_months*, *select_years*, *select_seasons*, and *select_timesteps*, with the args used being *variables*, *months*, *years*, *seasons*, and *timesteps*. Subsetting will occur in the order given. For example, if you want to select the years 1990 and 1991 and months June and July, you would do the following:

```
data.select(years = [1990, 1991], months = [6, 7])
```

2.12.74 select_months

This method allows you to subset a dataset to specific months. This can either be a single month, a list of months or a range. For example, if we wanted the first half of a year, we would do the following:

```
data.select_months(range(1, 7))
```

2.12.75 select_variables

This method allows you to subset a dataset to specific variables. This either accepts a single variable or a list of variables. For example, if you wanted two variables, var1 and var2, you would do the following:

```
data.select_variables(["var1", "var2"])
```

2.12.76 select_years

This method subsets datasets to specified years. It will accept either a single year, a list of years, or a range. For example, if you wanted to subset a dataset the 1990s, you would do the following:

```
data.select_years(range(1990, 2000))
```

2.12.77 set_missing

This method allows you to set a range to missing values. It either accepts a single variable or two variables, specifying the range to be set to missing values. For example, if you wanted all values between 0 and 10 to be set to missing, you would do the following:

```
data.set_missing([0, 10])
```

2.12.78 shift_days

This method allows you to shift time by a set number of hours, days, months or years. This acts as a wrapper for *shift_hours*, *shift_days*, *shift_months* and *shift_years*. Use the args *hours*, *days*, *months*, or *years*. This takes any number of arguments. So, if you wanted to shift time forward by 1 year, 1 month and 1 days you would do the following:

```
data.shift(years = 1, months = 1, days = 1)
```

2.12.79 shift_days

This method allows you to shift time by a set number of days. For example, if you want time moved forward by 2 hours you would do the following:

```
data.shift_days(2)
```

2.12.80 shift_hours

This method allows you to shift time by a set number of hours. For example, if you want time moved back by 1 hour you would do the following:

```
data.shift_hours(-1)
```

2.12.81 shift_months

This method allows you to shift time by a set number of months. For example, if you want time moved back by 2 months you would do the following:

```
data.shift_months(2)
```

2.12.82 shift_years

This method allows you to shift time by a set number of years. For example, if you want time moved back by 10 years you would do the following:

```
data.shift_years(10)
```

2.12.83 spatial_max

This method will calculate the maximum value observed in space for each variable and time step. Usage is simple:

```
data.spatial_max()
```

2.12.84 spatial_mean

This method will calculate the spatial mean for each variable and time step. If the grid cell area can be calculated, this will be an area weighted mean. Usage is simple:

```
data.spatial_mean()
```

2.12.85 spatial_min

This method will calculate the minimum observed in space for each variable and time step. Usage is simple:

```
data.spatial_min()
```

2.12.86 spatial_percentile

This method will calculate the percentile of variable across space for time step. For example, if you wanted to calculate the 75th percentile, you would do the following:

```
data.spatial_percentile(p=75)
```

2.12.87 spatial_range

This method will calculate the value range observed in space for each variable and time step. Usage is simple:

```
data.spatial_range()
```

2.12.88 spatial_sum

This method will calculate the spatial sum for each variable and time step. In some cases, for example when variables are concentrations, it makes more sense to multiply the value in each grid cell by the grid cell area, when doing a spatial sum. This method therefore has an argument by_area which defines whether to multiply the variable value by the area when doing the sum. By default by_area is False.

Usage is simple:

```
data.spatial_sum()
```

2.12.89 split

Except for methods that begin with merge or ensemble, all nctoolkit methods operate on individual files within a dataset. There are therefore cases when you might want to be able to split a dataset into separate files for analysis. This can be done using split, which let's you split a file into separate years, months or year/month combinations. For example, if you want to split a dataset into files of different years, you can do this:

```
data.split("year")
```

2.12.90 subtract

This method can subtract from a dataset. You can substract a constant, another dataset or a NetCDF file. In the case of datasets or NetCDF files the grids etc. must be of the same structure as the original dataset.

For example, if we had a temperature dataset where temperature was in Kelvin, we could convert it to Celsiu by subtracting 273.15.

```
data.subtract(273.15)
```

2.12.91 sum

This method will calculate the sum of values of all variables in all grid cells. Usage is simple:

```
data.sum()
```

sum_all —

This method will calculate the sum of all variables separately for each time cell and grid cell. Usage is simple:

```
data.sum_all()
```

2.12.92 surface

This method will extract the surface level from a multi-level dataset. Usage is simple:

```
data.surface()
```

2.12.93 to_dataframe

This method will return a pandas dataframe with the contents of the dataset. This has a decode_times argument to specify whether you want the times to be decoded. Defaults to True. Usage is simple:

```
data.to_dataframe()
```

2.12.94 to_latlon

This method will regrid a dataset to a regular latlon grid. The minimum and maximum longitudes and latitudes must be specified, along with the horizontal and vertical resolutions.

```
data.to_latlon(lon = [-80, 20], lat = [30, 80], res = [1,1])
```

2.12.95 to_xarray

This method will return an xarray datasetwith the contents of the dataset. This has a decode_times argument to specify whether you want the times to be decoded. Defaults to True. Usage is simple:

```
data.to_xarray()
```

2.12.96 transmute

This method can be used to generate new variables using arithmetic expressions. Existing will be removed from the dataset. See mutate if you want to keep existing variables. The method requires a dictionary, where the key-value pairs are the new variables and expression required to generate it.

For example, if had a temperature dataset, with temperature in Celsius, we might want to convert that to Kelvin. We can do this easily:

```
data.transmute({"temperature_k":"temperature+273.15"})
```

2.12.97 var

This method calculates the variance of each variable in the dataset. This is calculate across all time steps. Usage is simple:

```
data.var()
```

2.12.98 vertical_interp

This method interpolates variables vertically. It requires a list of vertical levels, for example depths, you want to interpolate. For example, if you had an ocean dataset and you wanted to interpolate to 10 and 20 metres you would do the following:

```
data.vertical_interp(levels = [10, 20])
```

2.12.99 vertical_max

This method calculates the maximum value of each variable across all vertical levels. Usage is simple:

```
data.vertical_max()
```

2.12.100 vertical_mean

This method calculates the mean value of each variable across all vertical levels. Usage is simple:

```
data.vertical_mean()
```

2.12.101 vertical_min

This method calculates the minimum value of each variable across all vertical levels. Usage is simple:

```
data.vertical_min()
```

2.12.102 vertical_range

This method calculates the value range of each variable across all vertical levels. Usage is simple:

```
data.vertical_range()
```

2.12.103 vertical_sum

This method calculates the sum each variable across all vertical levels. Usage is simple:

```
data.vertical_sum()
```

2.12.104 to_nc

This method allows you to write the contents of a dataset to a NetCDF file. If the target file exists and you want to overwrite it set overwrite to True. Usage is simple:

```
data.to_nc(outfile)
```

2.12.105 zip

This method will zip the contents of a dataset. This is mostly useful for processing chains where you want to minimize disk space usage by the output. Please note this method works lazily. In the code below only one file is generated, a zipped "outfile".

```
nc.options(lazy = True)
data = nc.open_data(infile)
data.select_years(1990)
data.zip()
data.write_nc(outfile)
```

2.12.106 zonal statistics

Calculate the following zonal statistics: mean, min, max and range:

```
data.zonal_mean()
data.zonal_min()
data.zonal_max()
data.zonal_range()
```

2.13 How to guide

This guide will show how to carry out key nctoolkit operations. We will use a sea surface temperature data set and a depth-resolved ocean temperature data set. The data set can be downloaded from here.

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

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

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

2.13.1 How to select years and months

If we want to select specific years and months we can use the `select_years` and `select_months` method

[2]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.select_years(1960)
sst.select_months(1)
sst.times
```

[2]:
```
['1960-01-01T00:00:00']
```

2.13.2 How to mean, mean, max etc.

If you want to calculate the mean value of a variable over all time steps you can use `mean`:

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

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

Similarly, if you want to calculate the minimum, maximum, sum and range of values over time just use `min`, `max`, `sum` and `range`.

2.13.3 How to copy a data set

If you want to make a deep copy of a data set, use the built in copy method. This method will return a new data set. This method should be used because of nctoolkit's built in methods to automatically delete temporary files that are no longer required. Behind the scenes, using copy will result in nctoolkit registering that it needs the NetCDF file for both the original dataset and the new copied one. So if you copy a dataset, and then delete the original, nctoolkit knows to not remove any NetCDF files related to the dataset.

[4]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.select_years(1960)
sst.select_months(1)
sst1 = sst.copy()
del sst
os.path.exists(sst1.current)
```

[4]:
```
True
```

2.13.4 How to clip to a region

If you want to clip the data to a specific longitude and latitude box, we can use `clip`, with the longitude and latitude range given by lon and lat.

[5]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.select_months(1)
sst.select_years(1980)
sst.clip(lon = [-80, 20], lat = [40, 70])
sst.plot()
```

[5]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

2.13.5 How to rename a variable

If we want to rename a variable we use the `rename` method, and supply a dictionary where the key-value pairs are the original and new names

[6]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.variables
```

[6]:
```
['sst']
```

The original dataset had only one variable called sst. We can now rename it, and display the new variables.

[7]:
```
sst.rename({"sst": "temperature"})
sst.variables
```

[7]:
```
['temperature']
```

2.13.6 How to create new variables

New variables can be created using arithmetic operations using either `mutate` or `transmute`. The `mutate` method will maintain the original variables, whereas `transmute` will not. This method requires a dictionary, where the key, values pairs are the names of the new variables and the arithemtic operations to perform. The example below shows how to create a new variable with

[8]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.mutate({"sst_k": "sst+273.15"})
sst.variables
```

[8]:
```
['sst', 'sst_k']
```

2.13.7 How to calculate a spatial average

You can calculate a spatial average using the `spatial_mean` method. There are additional methods for maximums etc.

[9]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.spatial_mean()
sst.plot()
```

[9]:
```
:Curve   [time]   (x)
```

2.13.8 How to calculate an annual mean

You can calculate an annual mean using the `annual_mean` method.

[10]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.spatial_mean()
sst.annual_mean()
sst.plot()
```

[10]:
```
:Curve   [time]   (x)
```

2.13.9 How to calculate a rolling average

You can calculate a rolling mean using the `rolling_mean` method, with the window argument providing the number of time steps to average over. There are additional methods for rolling sums etc. The code below will calculate a rolling mean of global SST using a 20 year window.

[11]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.spatial_mean()
sst.annual_mean()
sst.rolling_mean(20)
sst.plot()
```

[11]:
```
:Curve   [time]   (x)
```

2.13.10 How to calculate temporal anomalies

You can calculate annual temporal anomalies using the `annual_anomaly` method. This requires a baseline period.

[12]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.spatial_mean()
sst.annual_anomaly(baseline = [1960, 1979])
sst.plot()
```

[12]:
```
:Curve   [time]   (x)
```

2.13.11 How to split data by year etc

Files within a dataset can be split by year, day, year and month or season using the `split` method. If we wanted to split by year, we do the following:

[13]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.split("year")
```

2.13.12 How to merge files in time

We can merge files based on time using `merge_time`. We can do this by merging the dataset that results from splitting the original sst dataset. If we split the dataset by year, we see that there are 169 files, one for each year.

[14]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.split("year")
```

We can then merge them together to get a single file dataset:

[15]:
```
sst.merge_time()
```

2.13.13 How to do variables-based merging

If we have two more more files that have the same time steps, but different variables, we can merge them using `merge`. The code below will first create a dataset with a NetCDF file with SST in K, and it will then create a new dataset with this netcd file and the original, and then merge them.

[16]:
```
sst1 = nc.open_data("sst.mon.mean.nc")
sst2 = nc.open_data("sst.mon.mean.nc")
sst2.transmute({"sst_k": "sst+273.15"})
new_sst = nc.open_data([sst1.current, sst2.current])
new_sst.current
new_sst.merge()
```

In some cases we will have two or more datasets we want to merge. In this case we can use the `merge` function as follows:

[17]:
```
sst1 = nc.open_data("sst.mon.mean.nc")
sst2 = nc.open_data("sst.mon.mean.nc")
sst2.transmute({"sst_k": "sst+273.15"})
new_sst = nc.merge(sst1, sst2)
new_sst.variables
```

[17]:
```
['sst', 'sst_k']
```

2.13.14 How to horizontally regrid data

Variables can be regridded horizontally using `regrid`. This method requires the new grid to be defined. This can either be a pandas data frame, with lon/lat as columns, an xarray object, a NetCDF file or an nctolkit dataset. I will demonstrate all three methods by regridding SST to the North Atlantic. Let's begin by getting a grid for the North Atlantic.

[18]:
```
new_grid = nc.open_data("sst.mon.mean.nc")
new_grid.clip(lon = [-80, 20], lat = [30, 70])
new_grid.select_months(1)
new_grid.select_years( 2000)
```

First, we will use the new dataset itself to do the regridding. I will calculate mean SST using the original data, and then regrid to the North Atlantic.

[19]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.mean()
sst.regrid(grid = new_grid)
sst.plot()
```

[19]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

We can also do this using the NetCDF, which is new_grid.current

[20]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.mean()
sst.regrid(grid = new_grid.current)
sst.plot()
```

[20]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

or we can use a pandas data frame. In this case I will convert the xarray data set to a data frame.

[21]:
```
na_grid = xr.open_dataset(new_grid.current)
na_grid = na_grid.to_dataframe().reset_index().loc[:,["lon", "lat"]]
sst = nc.open_data("sst.mon.mean.nc")
sst.mean()
sst.regrid(grid = na_grid)
sst.plot()
```

[21]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

2.13.15 How to temporally interpolate

Temporal interpolation can be carried out using `time_interp`. This method requires a start date (start) of the format YYYY/MM/DD and an end date (end), and a temporal resolution (resolution), which is either 1 day ("daily"), 1 week ("weekly"), 1 month ("monthly"), or 1 year ("yearly").

[22]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.time_interp(start = "1990/01/01", end = "1990/12/31", resolution = "daily")
```

2.13.16 How to calculate a monthly average from daily data

If you have daily data, you can calculate a month average using `monthly_mean`. There are also methods for maximums etc.

[23]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.time_interp(start = "1990/01/01", end = "1990/12/31", resolution = "daily")
sst.monthly_mean()
```

2.13.17 How to calculate a monthly climatology

If we want to calculate the mean value of variables for each month in a given dataset, we can use the `monthly_mean_climatology` method as follows:

[24]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.monthly_mean_climatology()
sst.select_months(1)
sst.plot()
```

[24]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

2.13.18 How to calculate a seasonal climatology

[25]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.seasonal_mean_climatology()
sst.select_timesteps(0)
sst.plot()
```

[25]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

[26]:
```
## How to read a dataset using pandas or xarray
```

To read the dataset to an xarray Dataset use `to_xarray`:

[27]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.to_xarray()
```

[27]:
```
<xarray.Dataset>
Dimensions:  (lat: 180, lon: 360, time: 2028)
Coordinates:
  * lat      (lat) float32 89.5 88.5 87.5 86.5 85.5 ... -86.5 -87.5 -88.5 -89.5
  * lon      (lon) float32 0.5 1.5 2.5 3.5 4.5 ... 355.5 356.5 357.5 358.5 359.5
  * time     (time) datetime64[ns] 1850-01-01 1850-02-01 ... 2018-12-01
Data variables:
    sst      (time, lat, lon) float32 ...
Attributes:
    title:            created 12/2013 from data provided by JRA
    history:          Created 12/2012 from data obtained from JRA by ESRL/PSD
    platform:         Analyses
    citation:         Hirahara, S., Ishii, M., and Y. Fukuda,2014: Centennial...
    institution:      NOAA ESRL/PSD
    Conventions:      CF-1.2
    References:       http://www.esrl.noaa.gov/psd/data/gridded/cobe2.html
    dataset_title:    COBE-SST2 Sea Surface Temperature and Ice
    original_source:  https://climate.mri-jma.go.jp/pub/ocean/cobe-sst2/
```

To read the dataset in as a pandas dataframe use `to_dataframe`:

[28]:
```
sst.to_dataframe()
```

[28]:
```
                           sst
lat   lon   time
 89.5 0.5   1850-01-01 -1.712
            1850-02-01 -1.698
            1850-03-01 -1.707
            1850-04-01 -1.742
            1850-05-01 -1.725
...                      ...
-89.5 359.5 2018-08-01    NaN
            2018-09-01    NaN
            2018-10-01    NaN
            2018-11-01    NaN
            2018-12-01    NaN
```

(continues on next page)

(continued from previous page)

```
[131414400 rows x 1 columns]
```

2.13.19 How to calculate cell areas

If we want to calculate the area of each cell in a dataset, we use the `cell_area` method. The `join` argument let's you choose whether to join the cell areas to the existing dataset, or to only include cell areas in the dataset.

[29]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.cell_areas(join=False)
sst.plot()
```

[29]:
```
:Image   [lon,lat]   (cell_area)
```

2.13.20 How to use urls

If a file is located at a url, we can send it to `open_data`:

[30]:
```
url = "ftp://ftp.cdc.noaa.gov/Datasets/COBE2/sst.mon.ltm.1981-2010.nc"
sst = nc.open_data(url)
```

```
Downloading ftp://ftp.cdc.noaa.gov/Datasets/COBE2/sst.mon.ltm.1981-2010.nc
```

This will download the file from the url and save it as a temp file. We can then work with it as usual. A future release of nctoolkit will have thredds support.

2.13.21 How to calculate an ensemble average

nctoolkit has built in methods for working with ensembles. Let's start by splitting the 1850-2019 sst dataset into an ensemble, where each file is a separate year:

[31]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.split("year")
```

An ensemble mean can be calculated in two ways. First, we can calculate the mean in each time step. So here the files have temperature from 1850 onwards. We can calculate the monthly mean temperature over that time period as follows, and from there we can calculate the global mean:

[32]:
```
sst.ensemble_mean()
sst.spatial_mean()
sst.plot()
```

[32]:
```
:Curve   [time]   (x)
```

We might want to calculate the average over all time steps, i.e. calculating mean temperature since 1850. We do this by changing the `ignore_time` argument:

[33]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.split("year")
sst.ensemble_mean(ignore_time=True)
sst.plot()
```

[33]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

2.14 API Reference

2.14.1 Session options

options(**kwargs)	Define session options.

nctoolkit.options

nctoolkit.**options**(***kwargs*)

Define session options. Set the options in the session. Available options are thread_safe and lazy. Set thread_safe = True if hdf5 was built to be thread safe. Set lazy = 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")
```

2.14.2 Reading/copying data

open_data([x, suppress_messages, 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

nctoolkit.open_data

nctoolkit.**open_data**(*x=[]*, *suppress_messages=False*, *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.
- **thredds** (*boolean*) – Are you accessing a thredds server? Must end with .nc.
- **checks** (*boolean*) – Do you want basic checks to ensure cdo can read files?

Examples

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

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

nctoolkit.open_url

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

Read netcdf data from a url as a DataSet object

Parameters

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

Examples

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

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

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

nctoolkit.open_thredds

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

Read thredds data as a DataSet object

Parameters

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

Examples

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

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

nctoolkit.DataSet.copy

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

Make a deep copy of an DataSet object

2.14.3 Merging or analyzing multiple datasets

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

nctoolkit.merge

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

Merge datasets

Parameters

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

nctoolkit.cor_time

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

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

Parameters

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

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

2.14.4 Adding file(s) to a dataset

append

nctoolkit.append

Functions

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

2.14.5 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

nctoolkit.DataSet.variables

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

nctoolkit.DataSet.years

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

nctoolkit.DataSet.months

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

nctoolkit.DataSet.times

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

nctoolkit.DataSet.levels

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

nctoolkit.DataSet.size

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

nctoolkit.DataSet.current

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

nctoolkit.DataSet.history

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

nctoolkit.DataSet.start

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

2.14.6 Plotting

DataSet.plot(self[, vars])	

nctoolkit.DataSet.plot

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

2.14.7 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

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.

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:

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

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.

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?

2.14.8 NetCDF file attribute modification

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

nctoolkit.DataSet.set_longnames

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

Set the long names of variables

Parameters **name_dict** (*dict*) – Dictionary with key, value pairs representing the variable names and their long names

nctoolkit.DataSet.set_units

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

Set the units for variables

Parameters **unit_dict** (*dict*) – A dictionary where the key-value pairs are the variables and new units respectively.

2.14.9 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.

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:

```
>>> data.surface()
```

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

nctoolkit.DataSet.bottom

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

Extract the bottom level from a dataset This extracts the bottom level from each NetCDF file. Please note that for ensembles, it uses the first file to derive the index of the bottom level. Use bottom_mask for files when the bottom cell in NetCDF files do not represent the actual bottom.

Examples

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

```
>>> data.bottom()
```

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

nctoolkit.DataSet.vertical_interp

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

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:

```
>>> data.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.

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:

```
>>> data.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.

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:

```
>>> data.vertical_min()
```

nctoolkit.DataSet.vertical_max

DataSet.**vertical_max**(*self*)
Calculate the vertical maximum of variable values This is calculated for each time step and grid cell

Examples

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

```
>>> data.vertical_max()
```

nctoolkit.DataSet.vertical_range

DataSet.**vertical_range**(*self*)
Calculate the vertical range of variable values This is calculated for each time step and grid cell

Examples

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

```
>>> data.vertical_range()
```

nctoolkit.DataSet.vertical_sum

DataSet.**vertical_sum**(*self*)

Calculate the vertical sum of variable values This is calculated for each time step and grid cell

Examples

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

```
>>> data.vertical_sum()
```

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:

```
>>> data.vertical_sum()
```

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

nctoolkit.DataSet.invert_levels

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

Invert the levels of 3D variables This is calculated for each time step and grid cell

Examples

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

```
>>> data.invert_levels()
```

nctoolkit.DataSet.bottom_mask

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

Create a mask identifying the deepest cell without missing values. This converts a dataset to a mask identifying which cell represents the bottom, for example the seabed. 1 identifies the deepest cell with non-missing values. Everything else is 0, or missing. At present this method only uses the first available variable from netcdf files, so it may not be suitable for all data

2.14.10 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

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:

```
>>> data.rolling_mean(10)
```

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:

```
>>> data.rolling_min(10)
```

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:

```
>>> data.rolling_max(10)
```

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:

```
>>> data.rolling_sum(10)
```

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:

```
>>> data.rolling_range(10)
```

2.14.11 Evaluation setting

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

nctoolkit.DataSet.run

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

Examples

If evaluation is lazy and you need to evaluate commands on a dataset, do the following:

```
>>> data.run()
```

2.14.12 Cleaning functions

2.14.13 Ensemble creation

create_ensemble([path, recursive])	Generate an ensemble

nctoolkit.create_ensemble

nctoolkit.**create_ensemble**(*path=''*, *recursive=True*)
Generate an ensemble

Parameters

- **path** (*str*) – The directory to search for netcdf files
- **recursive** (*boolean*) – True/False depending on whether you want to search the path recursively. Defaults to True.

Returns A list of files

Return type list

Examples

If you wanted to recursively find all NetCDF files available in a directory "data", you would do this:

```
>>> import nctoolkit as nc
>>> nc.create_ensemble("data")
```

If you wanted to find the files in that directory and ignore subdirectories, you would instead do this:

```
>>> nc.create_ensemble("data", recursive = False)
```

2.14.14 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.

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:

```
>>> data.add(10)
```

To add the values in a dataset data2 from a dataset data1, you would do the following:

```
>>> data1.add(data2)
```

Grids in the datasets must match. Addition will occur in matching timesteps in data1 and data2. If there is only 1 timestep in data2, then the data from that timestep will be added to the data in all data1 time steps.

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

```
>>> data1.add("example.nc")
```

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:

```
>>> data.subtract(10)
```

To substract the values in a dataset data2 from those in a dataset data1, you would do the following:

```
>>> data1.subtract(data2)
```

Grids in the datasets must match. Division will occur in matching timesteps in data1 and data2 if there are matching timesteps. If there is only 1 timestep in data2, then the data from that timestep in data2 will be subtracted from the data in all timesteps in data1.

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

```
>>> data1.subtract("example.nc")
```

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:

```
>>> data.multiply(10)
```

To multiply the values in a dataset by the values of variables in dataset data2, you would do the following:

```
>>> data1.multiply(data2)
```

Grids in the datasets must match. Multiplication will occur in matching timesteps in data1 and data2. If there is only 1 timestep in data2, then the data from that timestep in data2 will multiply the data in all timesteps in data1.

Multiplying a dataset by the data from another NetCDF file will work in the same way:

```
>>> data1.multiply("example.nc")
```

nctoolkit.DataSet.divide

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

Divide the data This will divide the dataset by a constant, another dataset or a NetCDF file. :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:

```
>>> data.divide(10)
```

To divide values in a dataset by those in the dataset data2 from a dataset data1, you would do the following:

```
>>> data1.divide(data2)
```

Grids in the datasets must match. Division will occur in matching timesteps in data1 and data2. If there is only 1 timestep in data2, then the data from that timeste in data2 will divided the data in all data1 time steps.

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

```
>>> data1.divide("example.nc")
```

2.14.15 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.

nctoolkit.DataSet.ensemble_mean

DataSet.**ensemble_mean**(*self*, *nco=False*, *ignore_time=False*)

Calculate an ensemble mean

Parameters

- **nco** (*boolean*) – Do you want to use NCO for the calculation? Default is False, i.e. CDO is used. Modify default if run time is an issue.
- **ignore_time** (*boolean*) – If True the mean is calculated over all time steps. If False, the ensemble mean is calculated for each time steps; for example, if the ensemble is made up of monthly files the mean for each month will be calculated.

nctoolkit.DataSet.ensemble_min

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

Calculate an ensemble min

Parameters

- **nco** (*boolean*) – Do you want to use NCO for the calculation? Default is False, i.e. CDO is used. Modify default if run time is an issue.
- **ignore_time** (*boolean*) – If True the min is calculated over all time steps. If False, the ensemble min is calculated for each time steps; for example, if the ensemble is made up of monthly files the min for each month will be calculated.

nctoolkit.DataSet.ensemble_max

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

Calculate an ensemble maximum

Parameters

- **nco** (*boolean*) – Do you want to use NCO for the calculation? Default is False, i.e. CDO is used. Modify default if run time is an issue.
- **ignore_time** (*boolean*) – If True the max is calculated over all time steps. If False, the ensemble max is calculated for each time steps; for example, if the ensemble is made up of monthly files the max for each month will be calculated.

nctoolkit.DataSet.ensemble_percentile

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

Calculate an ensemble percentile This will calculate the 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.

nctoolkit.DataSet.ensemble_range

DataSet.**ensemble_range**(*self*)

Calculate an ensemble range The range is calculated for each time step; for example, if each file in the ensemble has 12 months of data the statistic will be calculated for each month.

nctoolkit.DataSet.ensemble_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.

2.14.16 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.

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:

```
>>> data.crop(lon = [-40, 30], lat = [-10, 40])
```

If you wanted to select only the northern hemisphere, the following will work:

```
>>> data.crop(lat = [0, 90])
```

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:

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

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

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

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

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

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

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

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

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

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

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

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:

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

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

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

2.14.17 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.

nctoolkit.DataSet.set_date

DataSet.**set_date**(*self*, *year=None*, *month=None*, *day=None*, *base_year=1900*)

Set the date in a dataset You should only do this if you have to fix/change a dataset with a single, not multiple dates.

Parameters

- **year** (*int*) – The year
- **month** (*int*) – The month
- **day** (*int*) – The day
- **base_year** (*int*) – The base year for time creation in the netcdf. Defaults to 1900.

nctoolkit.DataSet.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:

```
>>> data.shift(hours = -1)
```

If you wanted to shift all times forward 2 days, you would do the following:

```
>>> data.shift(days = 2)
```

If you wanted to shift all times forward 6 months, you would do the following:

```
>>> data.shift(months = 6)
```

If you wanted to shift all times forward 1 year, you would do the following:

```
>>> data.shift(years = 1)
```

This method will allow partial matches in arguments. So the following will do the same thing:

```
>>> data.shift(year = 2)
```

```
>>> data.shift(years = 2)
```

2.14.18 Interpolation and resampling methods

DataSet.regrid(self[, grid, method])	Regrid a dataset to a target grid
DataSet.to_latlon(self[, lon, lat, res, method])	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

nctoolkit.DataSet.regrid

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

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"; "bic" - "bicubic interpolation".

nctoolkit.DataSet.to_latlon

DataSet.**to_latlon**(*self*, *lon=None*, *lat=None*, *res=None*, *method='bil'*)

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". Bilinear: "bil"; Nearest neighbour: "nn".

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:

```
>>> data.resample_grid(2)
```

nctoolkit.DataSet.time_interp

DataSet.**time_interp**(*self*, *start=None*, *end=None*, *resolution='monthly'*)

Temporally interpolate variables based on date range and time resolution

Parameters

- **start** (*str*) – Start date for interpolation. Needs to be of the form YYYY/MM/DD or YYYY-MM-DD.
- **end** (*str*) – End date for interpolation. Needs to be of the form YYYY/MM/DD or YYYY-MM-DD. If end is not given interpolation will be to the final available time in the dataset.
- **resolution** (*str*) – Time steps used for interpolation. Needs to be "daily", "weekly", "monthly" or "yearly". Defaults to monthly.

nctoolkit.DataSet.timestep_interp

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

Temporally interpolate a dataset to given number of time steps between existing time steps

Parameters **steps** (*int*) – Number of time steps to interpolate between existing time steps. For example, if you wanted to go from daily to hourly data you would set steps=24.

2.14.19 Masking methods

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

nctoolkit.DataSet.mask_box

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

Mask a lon/lat box

Parameters

- **lon** (*list*) – Longitude range to mask. Must be of the form: [lon_min, lon_max]
- **lat** (*list*) – Latitude range to mask. Must be of the form: [lat_min, lat_max]

2.14.20 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.
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.

continues on next page

Table 21 – continued from previous page

DataSet.meridonial_range(self)	Calculate the meridonial range for each year/month combination in files.

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:

```
>>> data.tmean()
```

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

```
>>> data.tmean("year")
```

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

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

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

```
>>> data.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:

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

A daily climatological mean would be the following:

```
>>> data.tmean( "day")
```

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:

```
>>> data.tmin()
```

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

```
>>> data.tmin("year")
```

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

```
>>> data.tmin("month")
```

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

```
>>> data.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:

```
>>> data.tmin( "month")
```

A daily climatological minimum would be the following:

```
>>> data.tmin( "day")
```

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:

```
>>> data.tmedian()
```

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

```
>>> data.tmedian("year")
```

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

```
>>> data.tmedian("month")
```

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

```
>>> data.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:

```
>>> data.tmedian( "month")
```

A daily climatological median would be the following:

```
>>> data.tmedian( "day")
```

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:

```
>>> data.tpercentile(20)
```

If you want to calculate the 20th percentile for each year in a dataset, do this:

```
>>> data.tpercentile(20)
```

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:

```
>>> data.tmax()
```

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

```
>>> data.tmax("year")
```

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

```
>>> data.tmax("month")
```

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

```
>>> data.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:

```
>>> data.tmax( "month")
```

A daily climatological maximum would be the following:

```
>>> data.tmax( "day")
```

nctoolkit.DataSet.tsum

DataSet.**tsum**(*self*, *over='time'*)
Calculate the temporal sum of all variables

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:

```
>>> data.trange()
```

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

```
>>> data.trange("year")
```

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

```
>>> data.trange("month")
```

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

```
>>> data.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:

```
>>> data.trange( "month")
```

A daily climatological range would be the following:

```
>>> data.trange( "day")
```

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:

```
>>> data.tvar()
```

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

```
>>> data.tvar("year")
```

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

```
>>> data.tvar("month")
```

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

```
>>> data.tvar(["year", "month"])
```

This method will also let you easily calculate climatologies. So, if you wanted to calculate a monthly climatological var, you would do this:

```
>>> data.tvar( "month")
```

A daily climatological variance would be the following:

```
>>> data.tvar( "day")
```

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:

```
>>> data.tstdev()
```

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

```
>>> data.tstdev("year")
```

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

```
>>> data.tstdev("month")
```

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

```
>>> data.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:

```
>>> data.tstdev("month")
```

A daily climatological standard deviation would be the following:

```
>>> data.tstdev("day")
```

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:

```
>>> data.tcumsum()
```

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:

```
>>> data.cor_space("x", "y")
```

The correlation coefficient will be calculated for each time step.

nctoolkit.DataSet.cor_time

DataSet.**cor_time**(*self*, *var1=None*, *var2=None*)

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

Parameters

- **var1** (*str*) – The first variable
- **var2** (*str*) – The second variable

Examples

If you wanted to calculate the temporal correlation coefficient between variables x and y in a dataset, you would do this:

```
>>> data.cor_space("x", "y")
```

The correlation coefficient will be calculated for each grid cell. This method will indicate how temporally correlated variables are in different spatial regions.

nctoolkit.DataSet.spatial_mean

DataSet.**spatial_mean**(*self*)

Calculate the area weighted spatial mean for all variables This is performed for each time step.

Examples

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

```
>>> data.spatial_mean()
```

Note that this calculation will calculate the average using weights based on each cell's area. If cell areas cannot be calculated, it will take a straight average, and a warning will say this.

nctoolkit.DataSet.spatial_min

DataSet.**spatial_min**(*self*)

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

Examples

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

```
>>> data.spatial_min()
```

nctoolkit.DataSet.spatial_max

DataSet.**spatial_max**(*self*)
Calculate the spatial maximum for all variables This is performed for each time step.

Examples

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

```
>>> data.spatial_max()
```

nctoolkit.DataSet.spatial_percentile

DataSet.**spatial_percentile**(*self*, *p=None*)
Calculate the spatial sum for all variables This is performed for each time step. :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:

```
>>> data.spatial_percentile(50)
```

nctoolkit.DataSet.spatial_range

DataSet.**spatial_range**(*self*)
Calculate the spatial range for all variables This is performed for each time step.

Examples

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

```
>>> data.spatial_max()
```

nctoolkit.DataSet.spatial_sum

DataSet.**spatial_sum**(*self*, *by_area=False*)
Calculate the spatial sum for all variables This is performed for each time step.

Parameters **by_area** (*boolean*) – Set to True if you want to multiply the values by the grid cell area before summing over space. Default is False.

Examples

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

```
>>> data.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:

```
>>> data.spatial_sum(by_area = True)
```

Each cell's value will be multiplied by the area of the cell (in square metres) prior to calculating the spatial sum.

nctoolkit.DataSet.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.

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:

```
>>> data.zonal_mean()
```

nctoolkit.DataSet.zonal_min

DataSet.**zonal_min**(*self*)

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

Examples

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

```
>>> data.zonal_min()
```

nctoolkit.DataSet.zonal_max

DataSet.**zonal_max**(*self*)

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

Examples

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

```
>>> data.zonal_max()
```

nctoolkit.DataSet.zonal_range

DataSet.**zonal_range**(*self*)

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

Examples

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

```
>>> data.zonal_range()
```

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:

```
>>> data.meridonial_mean()
```

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:

```
>>> data.meridonial_min()
```

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:

```
>>> data.meridonial_max()
```

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:

```
>>> data.meridonial_max()
```

2.14.21 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.

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.

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.

2.14.22 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.

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:

```
>>> data.split("year")
```

If you wanted to split by month, do the following:

```
>>> data.split("month")
```

2.14.23 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.

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:

```
>>> data.to_nc("out.nc")
```

By default this file will be zipped. If you do not want it zipped, do this:

```
>>> data.to_nc("out.nc", zip = False)
```

By default this cannot overwrite files. If the output file exists, do the following:

```
>>> data.to_nc("out.nc", overwrite = True)
```

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

Examples

If you want to convert a dataset to an xarray dataset, do the following:

```
>>> data.to_xarray()
```

This will return an xarray dataset.

If you do not want time to be decoded, do the following:

```
>>> data.to_xarray(decode_times = False)
```

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

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:

```
>>> data.zip()
```

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

nctoolkit.DataSet.format

DataSet.**format**(*self*, *ext=None*)
Zip the dataset This will compress the files within the dataset. This works lazily. :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

2.14.24 Miscellaneous methods

DataSet.cell_area(self[, join])	Calculate the area of grid cells.
DataSet.cdo_command(self[, command])	Apply a cdo command
DataSet.nco_command(self[, command, ensemble])	Apply an nco command
DataSet.compare_all(self[, expression])	Compare all variables to a constant
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.

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:

```
>>> data.cell_area()
```

If you wanted to replace a dataset with the cell areas of that dataset, you would do the following:

```
>>> data.cell_area(join = False)
```

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.

nctoolkit.DataSet.nco_command

DataSet.**nco_command**(*self*, *command=None*, *ensemble=False*)

Apply an nco command

Parameters

- **command** (*string*) – nco command to call. This must be of a form such that "nco {command} infile outfile" will run.
- **ensemble** (*boolean*) – Set to True if you want the command to take all of the files as input. This is useful for ensemble methods.

nctoolkit.DataSet.compare_all

DataSet.**compare_all**(*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:

```
>>> data.compare_all(">0")
```

This will be calculcated for each time step.

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

```
>>> data.compare_all("<0")
```

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:

```
>>> data.reduce_dims("out.nc")
```

Note that this will work lazily. This method is most useful when you want to simplify datasets before exporting them to something like a pandas dataframe.

nctoolkit.DataSet.reduce_grid

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

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

The mask must have an identical grid to the dataset.

2.14.25 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.

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.

2.15 Package info

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

2.15.1 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.

2.15.2 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

n

INDEX

S

T

V

Y

Z