Xarray Part 2

https://xarray.pydata.org/en/stable/index.html

```
import numpy as np
import xarray as xr
```

```
path = '../data/precip_day*.nc'
```

```
ds = xr.open_mfdataset(path)
da = ds.precip
ds
```

```
<xarray.Dataset>
Dimensions:  (time: 12, lon: 192, lat: 96)
Coordinates:
  * time     (time) datetime64[ns] 2001-01-01 ... 2001-01-03T18:00:00
  * lon      (lon) float64 -180.0 -178.1 -176.2 -174.4 ... 174.4 176.2 178.1
  * lat      (lat) float64 88.57 86.72 84.86 83.0 ... -83.0 -84.86 -86.72 -88.57
Data variables:
    precip   (time, lat, lon) float32 dask.array<chunksize=(4, 96, 192), meta=np.ndarray>
Attributes:
    CDI:          Climate Data Interface version 1.9.9 (https://mpimet.mpg.de...
    Conventions:  CF-1.6
    history:      Wed Jul 14 15:47:55 2021: cdo -splitday -selname,precip rec...
    CDO:          Climate Data Operators version 1.9.9 (https://mpimet.mpg.de...
```

```
ds['precip_new'] = ds['precip']
ds
```

```
<xarray.Dataset>
Dimensions:     (time: 12, lon: 192, lat: 96)
Coordinates:
  * time        (time) datetime64[ns] 2001-01-01 ... 2001-01-03T18:00:00
  * lon         (lon) float64 -180.0 -178.1 -176.2 -174.4 ... 174.4 176.2 178.1
  * lat         (lat) float64 88.57 86.72 84.86 83.0 ... -84.86 -86.72 -88.57
Data variables:
    precip      (time, lat, lon) float32 dask.array<chunksize=(4, 96, 192), meta=np.ndarray>
    precip_new  (time, lat, lon) float32 dask.array<chunksize=(4, 96, 192), meta=np.ndarray>
```

```
Attributes:
    CDI:          Climate Data Interface version 1.9.9 (https://mpimet.mpg.de...
    Conventions:  CF-1.6
    history:      Wed Jul 14 15:47:55 2021: cdo -splitday -selname,precip rec...
    CDO:          Climate Data Operators version 1.9.9 (https://mpimet.mpg.de...
```

Select data

From `xarray`'s Indexing and selecting.

In addition to `numpy`'s and Python's `[]` syntax (`array[i, j]`, where `i` and `j` are both integers), xarray allows selecting data by **name**. Objects can store coordinates corresponding to each dimension of an array so that *label-based* indexing is also possible. In label-based indexing, the element position `i` is automatically looked-up from the coordinate values.

Dimensions of `xarray` objects have names, so you can also lookup the dimensions by name, instead of remembering their positional order.

> `Label` based selections mean that we do not need to know *anything* about the shape of the data array but can still work with the data.

Thus in total, `xarray` supports four different kinds of indexing, as described below and summarized in this table:

Dimension lookup	Index lookup	DataArray syntax	Dataset syntax
Positional	By integer	`da[0, :, :]`	not available
Positional	By label	`da.loc["2001-01-01", :, :]`	not available
By name	By integer	`da.isel(time=0)` or `da[dict(time=0)]`	`ds.isel(time=0)` or `ds[dict(time=0)]`
By name	By label	`da.sel(time="2001-01-01")` or `da.loc[dict(time="2001-01-01")]`	`ds.sel(time="2001-01-01")` or `ds.loc[dict(time="2001-01-01")]`

```
da = ds.precip
```

```
#1:
da[0,:,:]
```

```
<xarray.DataArray 'precip' (lat: 96, lon: 192)>
dask.array<getitem, shape=(96, 192), dtype=float32, chunksize=(96, 192), chunktype=numpy.nda
Coordinates:
    time     datetime64[ns] 2001-01-01
  * lon      (lon) float64 -180.0 -178.1 -176.2 -174.4 ... 174.4 176.2 178.1
```

```
  * lat      (lat) float64 88.57 86.72 84.86 83.0 ... -83.0 -84.86 -86.72 -88.57
Attributes:
    long_name:      total precipitation
    units:          kg/m^2s
    code:           4
    table:          128
    CDI_grid_type:  gaussian
```

```
#2
da1 = da.loc["2001-01-01T00:00:00", : , :]
da2 = da.loc["2001-01-01T00:00:00"]
print((da1.values == da2.values).all())
```

```
True
```

```
#3.1
da.isel(time=0)
```

```
<xarray.DataArray 'precip' (lat: 96, lon: 192)>
dask.array<getitem, shape=(96, 192), dtype=float32, chunksize=(96, 192), chunktype=numpy.nda
Coordinates:
    time     datetime64[ns] 2001-01-01
  * lon      (lon) float64 -180.0 -178.1 -176.2 -174.4 ... 174.4 176.2 178.1
  * lat      (lat) float64 88.57 86.72 84.86 83.0 ... -83.0 -84.86 -86.72 -88.57
Attributes:
    long_name:      total precipitation
    units:          kg/m^2s
    code:           4
    table:          128
    CDI_grid_type:  gaussian
```

```
#3.2
ds.isel(time=0)
```

```
<xarray.Dataset>
Dimensions:     (lon: 192, lat: 96)
Coordinates:
    time        datetime64[ns] 2001-01-01
  * lon         (lon) float64 -180.0 -178.1 -176.2 -174.4 ... 174.4 176.2 178.1
  * lat         (lat) float64 88.57 86.72 84.86 83.0 ... -84.86 -86.72 -88.57
Data variables:
    precip      (lat, lon) float32 dask.array<chunksize=(96, 192), meta=np.ndarray>
    precip_new  (lat, lon) float32 dask.array<chunksize=(96, 192), meta=np.ndarray>
Attributes:
    CDI:          Climate Data Interface version 1.9.9 (https://mpimet.mpg.de...
    Conventions:  CF-1.6
    history:      Wed Jul 14 15:47:55 2021: cdo -splitday -selname,precip rec...
    CDO:          Climate Data Operators version 1.9.9 (https://mpimet.mpg.de...
```

```
#4.1
da.sel(time="2001-01-01T00:00:00")
```

```
<xarray.DataArray 'precip' (lat: 96, lon: 192)>
dask.array<getitem, shape=(96, 192), dtype=float32, chunksize=(96, 192), chunktype=numpy.nda
Coordinates:
    time     datetime64[ns] 2001-01-01
  * lon      (lon) float64 -180.0 -178.1 -176.2 -174.4 ... 174.4 176.2 178.1
  * lat      (lat) float64 88.57 86.72 84.86 83.0 ... -83.0 -84.86 -86.72 -88.57
Attributes:
    long_name:      total precipitation
    units:          kg/m^2s
    code:           4
    table:          128
    CDI_grid_type:  gaussian
```

```
#4.2
ds.sel(time="2001-01-01T00:00:00")
```

```
<xarray.Dataset>
Dimensions:     (lon: 192, lat: 96)
Coordinates:
    time        datetime64[ns] 2001-01-01
  * lon         (lon) float64 -180.0 -178.1 -176.2 -174.4 ... 174.4 176.2 178.1
  * lat         (lat) float64 88.57 86.72 84.86 83.0 ... -84.86 -86.72 -88.57
Data variables:
    precip      (lat, lon) float32 dask.array<chunksize=(96, 192), meta=np.ndarray>
    precip_new  (lat, lon) float32 dask.array<chunksize=(96, 192), meta=np.ndarray>
Attributes:
    CDI:          Climate Data Interface version 1.9.9 (https://mpimet.mpg.de...
    Conventions:  CF-1.6
    history:      Wed Jul 14 15:47:55 2021: cdo -splitday -selname,precip rec...
    CDO:          Climate Data Operators version 1.9.9 (https://mpimet.mpg.de...
```

- Select **ranges** of data with `lower` and `upper` range values either by `numpy`-like access separated by `:` or with `.sel` and a `slice(lower, upper)`
- Find the **nearest** value for *label based* selection with keyword `method`, e.g. `method="nearest"`

```
ds.sel(time="2001-01-01T03:00:00", method='nearest').time.values
```

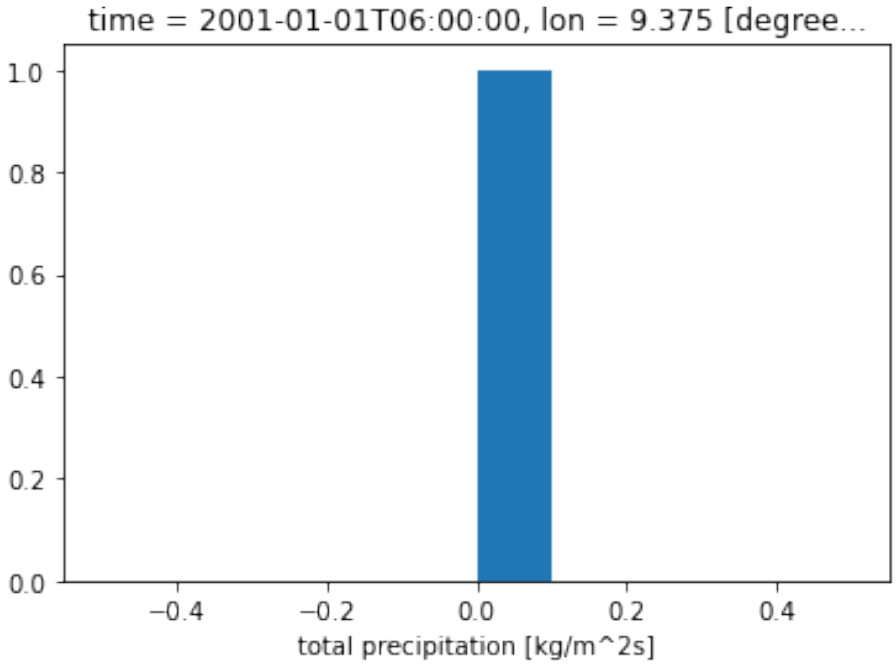
```
numpy.datetime64('2001-01-01T06:00:00.000000000')
```

Select the `pr` data for the grid box where Hamburg is in.

```
ds['precip'].sel(time="2001-01-01T03:00:00", lat=53., lon=10., method='nearest').plot()
```

```
(array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.]),
 array([-4.9999845e-01, -3.9999846e-01, -2.9999846e-01, -1.9999847e-01,
        -9.9998467e-02,  1.5369151e-06,  1.0000154e-01,  2.0000154e-01,
```

```
       3.0000153e-01,  4.0000153e-01,  5.0000155e-01], dtype=float32),
<BarContainer object of 10 artists>)
```



- **Masking** data with `where` allows to mask a data array with conditions on the *coordinates*. It applies on a *mask* of type `bool` with `True` and `False` values which we get by a condition. If we want to have all values of the southern hemisphere (`da.lat<0.`), we can do:

```
da.where(da.lat<0., drop=True)
```

where the argument `drop=True` removes the *False* values.

```
da.where(da.lat<0., drop=True)
```

```
<xarray.DataArray 'precip' (time: 12, lat: 48, lon: 192)>
dask.array<where, shape=(12, 48, 192), dtype=float32, chunksize=(4, 48, 192), chunktype=nump
Coordinates:
  * time     (time) datetime64[ns] 2001-01-01 ... 2001-01-03T18:00:00
  * lon      (lon) float64 -180.0 -178.1 -176.2 -174.4 ... 174.4 176.2 178.1
  * lat      (lat) float64 -0.9326 -2.798 -4.663 -6.528 ... -84.86 -86.72 -88.57
Attributes:
    long_name:      total precipitation
    units:          kg/m^2s
    code:           4
    table:          128
    CDI_grid_type:  gaussian
```

Computations (Xarray methods)

Xarray includes the scientific libraries of Python stack, Numpy and pandas. This means we can use their capabilities for computations:

```
da.max()
da.min()
da.std()
```

Converting units manually

The `precip` variable in the `ds` dataset has the units `kg m-2 s-1`. Lets assume that this is an average over the prior 6hours. In order to get the 6-hourly sum of this average, we need to multiply the data with a constant `c`. That can be done with

```
precipt_mon=da*c
```

1. Calculate the maximum of the daily precipitation sum of all the three files.
2. On what day did we have the highest daily precipitation?
3. Which grid point is associated with the highest precipitation value?

```
da_summax = da.sum().max()
da_summax.values
```

```
array(6.113039, dtype=float32)
```

```
da_max = da.max()
da_max.values
```

```
array(0.00186992, dtype=float32)
```

```
da_high = da.where(da == da_max.values, drop=True)
da_high.time.values[0]
```

```
numpy.datetime64('2001-01-03T18:00:00.000000000')
```

```
print(f'lon, lat = {da_high.lon.values[0]}, {da_high.lat.values[0]}')
```

```
lon, lat = 116.25, -15.854703869694877
```

Datetime

With the underlying `datetime64` library, `xarray` allows *derived* `datetime` components. With that, we can easily select temporal subsets from a dataset. The following commands return time values from your dataset.

```
ds["time.month"]
ds["time.dayofyear"]
ds["time.season"]
```

```
#is the same as:
ds["time"].dt.season
```

```
ds["time.month"]
```

```
<xarray.DataArray 'month' (time: 12)>
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
Coordinates:
  * time     (time) datetime64[ns] 2001-01-01 ... 2001-01-03T18:00:00
```

```
ds["time.dayofyear"]
```

```
<xarray.DataArray 'dayofyear' (time: 12)>
array([1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3])
Coordinates:
  * time     (time) datetime64[ns] 2001-01-01 ... 2001-01-03T18:00:00
```

```
ds["time.season"]
```

```
<xarray.DataArray 'season' (time: 12)>
array(['DJF', 'DJF', 'DJF', 'DJF', 'DJF', 'DJF', 'DJF', 'DJF', 'DJF',
       'DJF', 'DJF', 'DJF'], dtype='<U3')
Coordinates:
  * time     (time) datetime64[ns] 2001-01-01 ... 2001-01-03T18:00:00
```

```
#is the same as:
ds["time"].dt.season
```

```
<xarray.DataArray 'season' (time: 12)>
array(['DJF', 'DJF', 'DJF', 'DJF', 'DJF', 'DJF', 'DJF', 'DJF', 'DJF',
       'DJF', 'DJF', 'DJF'], dtype='<U3')
Coordinates:
  * time     (time) datetime64[ns] 2001-01-01 ... 2001-01-03T18:00:00
```

Resampling

For upsampling or downsampling temporal resolutions, `xarray` offers a `resample()` method which takes a target frequency as argument. For example, we can downsample our dataset from 6-hourly to daily:

```
ds.resample(time="1D")
```

This command analyzes the *time* values of the dataset and creates groups for the resampling frequency. In a next step, you specify **how to fill** the groups. The base time of the resampled dataset is the first time step of the original. If you only want the values from `00:00`, which is part of the first time value, you can use `nearest()`:

```
ds.resample(time="1D").nearest()
```

```
ds.resample(time="1D").nearest()
```

```
<xarray.Dataset>
Dimensions:     (time: 3, lon: 192, lat: 96)
Coordinates:
  * time        (time) datetime64[ns] 2001-01-01 2001-01-02 2001-01-03
  * lon         (lon) float64 -180.0 -178.1 -176.2 -174.4 ... 174.4 176.2 178.1
  * lat         (lat) float64 88.57 86.72 84.86 83.0 ... -84.86 -86.72 -88.57
Data variables:
    precip      (time, lat, lon) float32 dask.array<chunksize=(1, 96, 192), meta=np.ndarray>
    precip_new  (time, lat, lon) float32 dask.array<chunksize=(1, 96, 192), meta=np.ndarray>
Attributes:
    CDI:          Climate Data Interface version 1.9.9 (https://mpimet.mpg.de...
    Conventions:  CF-1.6
    history:      Wed Jul 14 15:47:55 2021: cdo -splitday -selname,precip rec...
    CDO:          Climate Data Operators version 1.9.9 (https://mpimet.mpg.de...
```

Again, it is important to check the *arguments* of these functions which allow you to configure in detail how to select the data.

If you would like to have the values from 06:00 am, you can set an *offset* as argument:

```
ds.resample(time="1D", base="6H").nearest()
```

```
ds.resample(time="1D", base="6H").nearest()
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
Input In [68], in <cell line: 1>()
----> 1 ds.resample(time="1D", base="6H").nearest()

File /sw/spack-levante/mambaforge-4.11.0-0-Linux-x86_64-sobz6z/lib/python3.9/site-packages/x
   1158         grouper = CFTimeGrouper(freq, closed, label, base, loffset)
   1159     else:
-> 1160         grouper = pd.Grouper(
   1161             freq=freq, closed=closed, label=label, base=base, loffset=loffset
   1162         )
   1163 group = DataArray(
   1164     dim_coord, coords=dim_coord.coords, dims=dim_coord.dims, name=RESAMPLE_DIM
   1165 )
   1166 resampler = self._resample_cls(
   1167     self,
   1168     group=group,
   (...)
   1172     restore_coord_dims=restore_coord_dims,
   1173 )

File /sw/spack-levante/mambaforge-4.11.0-0-Linux-x86_64-sobz6z/lib/python3.9/site-packages/p
```

```
   1554     raise ValueError("'offset' and 'base' cannot be present at the same time")
   1556 if base and isinstance(freq, Tick):
   1557     # this conversion handle the default behavior of base and the
   1558     # special case of GH #10530. Indeed in case when dealing with
   1559     # a TimedeltaIndex base was treated as a 'pure' offset even though
   1560     # the default behavior of base was equivalent of a modulo on
   1561     # freq_nanos.
-> 1562     self.offset = Timedelta(base * freq.nanos // freq.n)
   1564 if isinstance(loffset, str):
   1565     loffset = to_offset(loffset)

MemoryError:
```

GroupBy: split-apply-combine¶

How can we do a statistical **analysis** on that precipitation time series?

One *principle* of `xarray` operations (similar as pandas) is to implement the **split-apply-combine** strategy (from `xarray`'s doc).

1. Split your data into multiple independent groups with `groupby`.
2. Apply some function to each group, e.g. `mean`.
3. (Combine your groups back into a single data object if necessary).

We did it already in the *resample* section where we created groups with `resample` for frequencies and then *applied* a function to these groups. The *combine* is often implicated.

`groupby` operations work on both Dataset and DataArray objects. Let's see what that means to our `time` axis:

```
ds.groupby("time.dayofyear").mean()
```

```
<xarray.Dataset>
Dimensions:     (lon: 192, lat: 96, dayofyear: 3)
Coordinates:
  * lon         (lon) float64 -180.0 -178.1 -176.2 -174.4 ... 174.4 176.2 178.1
  * lat         (lat) float64 88.57 86.72 84.86 83.0 ... -84.86 -86.72 -88.57
  * dayofyear   (dayofyear) int64 1 2 3
Data variables:
    precip      (dayofyear, lat, lon) float32 dask.array<chunksize=(1, 96, 192), meta=np.nda
    precip_new  (dayofyear, lat, lon) float32 dask.array<chunksize=(1, 96, 192), meta=np.nda
```

We can also create groups with *ranges* which can gives us a histogram-like result.
E.g., grouping into latitude ranges of *30* degrees will look like:

```
 ds.groupby_bins("lat", [0, 30, 60, 90],
                 labels=["nequator", "nsubtropic", "nhighlats"]).mean()
```

```
<xarray.Dataset>
Dimensions:     (lat_bins: 3, time: 12, lon: 192)
Coordinates:
  * lat_bins    (lat_bins) object 'nequator' 'nsubtropic' 'nhighlats'
  * time        (time) datetime64[ns] 2001-01-01 ... 2001-01-03T18:00:00
  * lon         (lon) float64 -180.0 -178.1 -176.2 -174.4 ... 174.4 176.2 178.1
Data variables:
    precip      (lat_bins, time, lon) float32 dask.array<chunksize=(1, 4, 192), meta=np.ndar
    precip_new  (lat_bins, time, lon) float32 dask.array<chunksize=(1, 4, 192), meta=np.ndar
```

In which of 3 *climate zone* (assumed to have boardes at latitudes) occurs the highest average daily precip over the three given days?

1. Sum up the 6hourly sum of precipitation values to daily sum of precipation.
2. Average over `lon`
3. Apply `groupby_bins` and take negative latitudes into account.
4. Calculate the result

Write a Dataset to netCDF file

Xarray provides the functions `to_netcdf` or `to_zarr` to write a dataset to a file of these formats. The file ending for netCDF files is `.nc`.

```
precip_day.to_netcdf(<filename>)
```

Let's write our *daily* precipitation diagnostic to a new file and reopen it.

Working with missing values

- In observations, missing values are set in for gaps
- In model data, missing values are used when the model does not produce output for a cell

`xarray` treats missing values as `NaN`s, and there are different ways to handle missing values in Xarray. The methods `isnull`, `notnull`, `dropna`, `fillna`, and `count` can be used to work with data with missing values.

We now disucss 6 use cases abouthandling missing values with xarray.

In standardized netCDF files, the *missing value* of a variable is given as a variable attribute. But we will not always work with *standardized* files, so it is beneficial to know how to set the missing value of an array:

1. Set a value in an array to missing value

`numpy`'s `np.nan` method is used to define -9999 as the missing value.

```
tarray = xr.DataArray(data=[0, 1, -9999, 3, 4, 5, 6, 7, -9999, 9, 10], dims='x')
tarray = tarray.where(tarray != -9999, np.nan)

print(tarray)
```

Result:

```
<xarray.DataArray (x: 11)>
array([ 0.,  1., nan,  3.,  4.,  5.,  6.,  7., nan,  9., 10.])
Dimensions without coordinates: x
```

```
tarray = xr.DataArray(data=[0, 1, -9999, 3, 4, 5, 6, 7, -9999, 9, 10], dims='x')
tarray = tarray.where(tarray != -9999, np.nan)

print(tarray)
```

```
<xarray.DataArray (x: 11)>
array([ 0.,  1., nan,  3.,  4.,  5.,  6.,  7., nan,  9., 10.])
Dimensions without coordinates: x
```

1. Check where *missing values* exist. It returns a mask array of True/False elements.

```
print(tarray.isnull())
```

Result:

```
<xarray.DataArray (x: 11)>
array([False, False,  True, False, False, False, False, False,  True,
       False, False])
Dimensions without coordinates: x
```

```
print(tarray.isnull())
```

```
<xarray.DataArray (x: 11)>
array([False, False,  True, False, False, False, False, False,  True,
       False, False])
Dimensions without coordinates: x
```

1. The opposition: Set value in mask file to True when value is not NaN (missing value)

```
print(tarray.notnull())
```

Result:

```
<xarray.DataArray (x: 11)>
array([ True,  True, False,  True,  True,  True,  True,  True, False,
        True,  True])
Dimensions without coordinates: x
```

```
print(tarray.notnull())
```

```
<xarray.DataArray (x: 11)>
array([ True,  True, False,  True,  True,  True,  True,  True, False,
        True,  True])
Dimensions without coordinates: x
```

1. Count value that are not missing values.

```
print(tarray.count())
```

Result:

```
<xarray.DataArray ()>
array(9)
```

```
print(tarray.count())
```

```
<xarray.DataArray ()>
array(9)
```

1. *Drop* all `NaN`s of an array. Return all array elements that are not missing values.

```
print(tarray.dropna(dim='x'))
```

Result:

```
<xarray.DataArray (x: 9)>
array([ 0.,  1.,  3.,  4.,  5.,  6.,  7.,  9., 10.])
Dimensions without coordinates: x
```

```
print(tarray.dropna(dim='x'))
```

```
<xarray.DataArray (x: 9)>
array([ 0.,  1.,  3.,  4.,  5.,  6.,  7.,  9., 10.])
Dimensions without coordinates: x
```

1. Set missing value to a constant number

```
print(tarray.fillna(0))
```

Result:

```
<xarray.DataArray (x: 11)>
array([ 0.,  1.,  0.,  3.,  4.,  5.,  6.,  7.,  0.,  9., 10.])
Dimensions without coordinates: x
```

```
print(tarray.fillna(0))
```

```
<xarray.DataArray (x: 11)>
array([ 0.,  1.,  0.,  3.,  4.,  5.,  6.,  7.,  0.,  9., 10.])
Dimensions without coordinates: x
```

Reshaping

There are different ways to swap the dimensions of an array from (x,y) to (y,x), on the one hand the `transpose` and on the other hand the `T` methods.

Example:

```
B = xr.DataArray(np.arange(1, 31).reshape(6, 5), dims=('x', 'y'))
```

Result:

```
<xarray.DataArray (x: 6, y: 5)>
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25],
       [26, 27, 28, 29, 30]])
Dimensions without coordinates: x, y
```

Now, let us see what happens using the `transpose` method:

```
print(B.transpose())
```

Result:

```
<xarray.DataArray (y: 5, x: 6)>
array([[ 1,  6, 11, 16, 21, 26],
       [ 2,  7, 12, 17, 22, 27],
       [ 3,  8, 13, 18, 23, 28],
       [ 4,  9, 14, 19, 24, 29],
       [ 5, 10, 15, 20, 25, 30]])
Dimensions without coordinates: y, x
```

And the second way with the `T` method:

```
print(B.T)
```

Result:

```
<xarray.DataArray (y: 5, x: 6)>
array([[ 1,  6, 11, 16, 21, 26],
       [ 2,  7, 12, 17, 22, 27],
       [ 3,  8, 13, 18, 23, 28],
       [ 4,  9, 14, 19, 24, 29],
       [ 5, 10, 15, 20, 25, 30]])
Dimensions without coordinates: y, x
```

Create your own arrays and reshape them as you like.

```
B = xr.DataArray(np.arange(1, 31).reshape(6, 5), dims=('x', 'y'))
print(B.transpose())
print(B.T)
```

```
<xarray.DataArray (y: 5, x: 6)>
array([[ 1,  6, 11, 16, 21, 26],
       [ 2,  7, 12, 17, 22, 27],
       [ 3,  8, 13, 18, 23, 28],
       [ 4,  9, 14, 19, 24, 29],
       [ 5, 10, 15, 20, 25, 30]])
Dimensions without coordinates: y, x
<xarray.DataArray (y: 5, x: 6)>
array([[ 1,  6, 11, 16, 21, 26],
       [ 2,  7, 12, 17, 22, 27],
       [ 3,  8, 13, 18, 23, 28],
       [ 4,  9, 14, 19, 24, 29],
       [ 5, 10, 15, 20, 25, 30]])
Dimensions without coordinates: y, x
```

Plotting

Some additional examples demonstrate how to use the Xarray plot method.

```
ds=xr.open_dataset('../data/tsurf.nc')
```

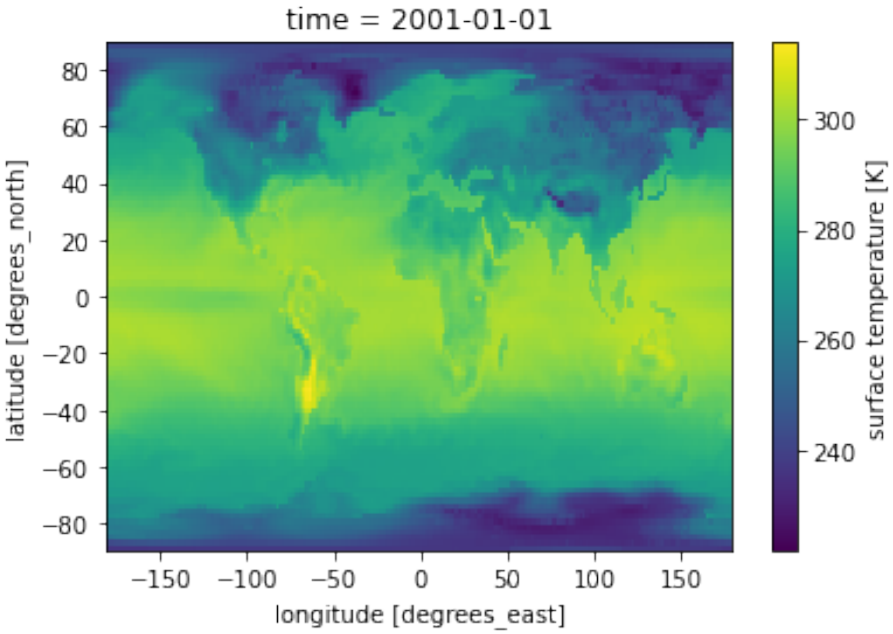
Example 1: Create the plot of the variable temperature (first timestep)

```
ds.temp[0,:,:].plot()
```

Note that the axes are annotated automatically.

```
ds.tsurf[0,:,:].plot()
```

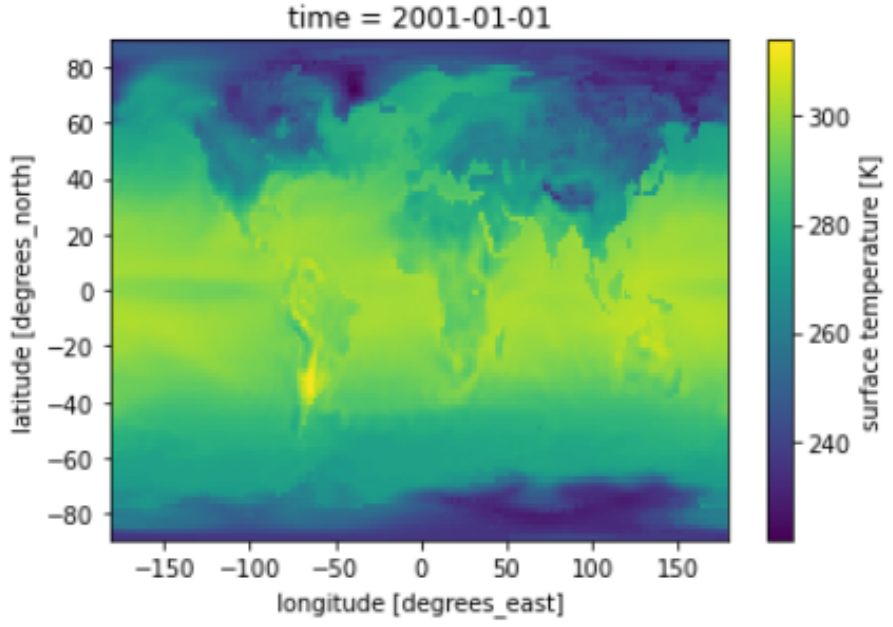
```
<matplotlib.collections.QuadMesh at 0x7fff67cb2070>
```



Example 2: Create the plot of the variable tsurf first timestep from file.

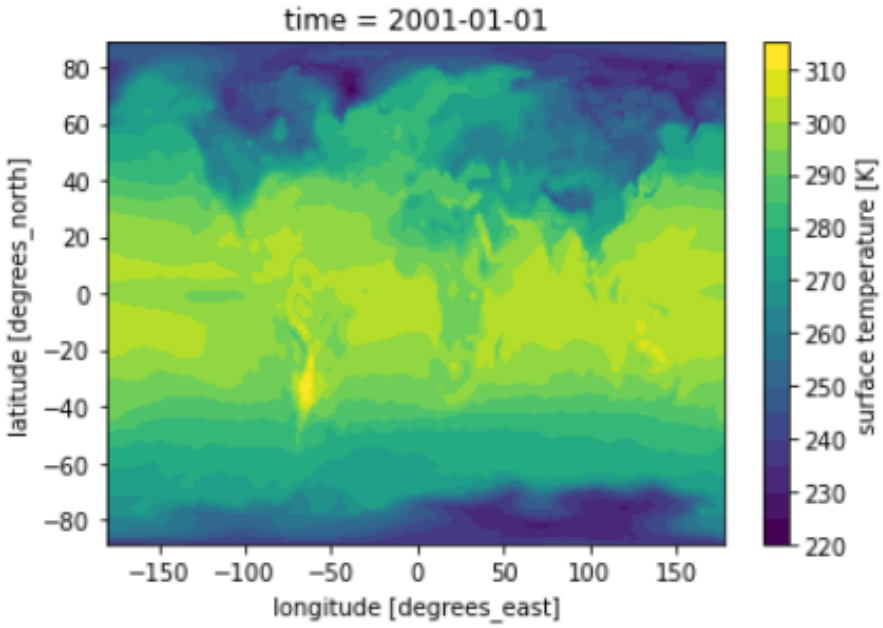
Create the plot using the default settings.

```
xr.open_dataset('../data/tsurf.nc').tsurf[0,:,:].plot()
```



Set the plot type to contourf (filled contours) and set the number of color intervals to 20.

```
xr.open_dataset('../data/tsurf.nc').tsurf[0,:,:].plot.contourf(levels=20)
```



1. Create the default plot
2. Create a filled contour plot with only 10 levels

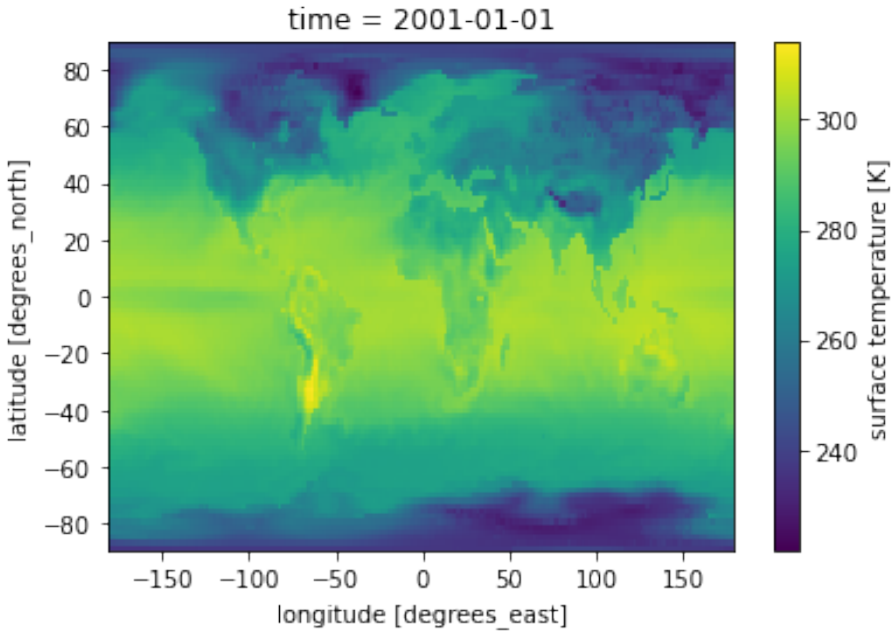
   Note: Run the commands in two different notebook cells.

1. What happens when you run it in the same notebook cell?

```
ds = xr.open_dataset('../data/tsurf.nc')
```

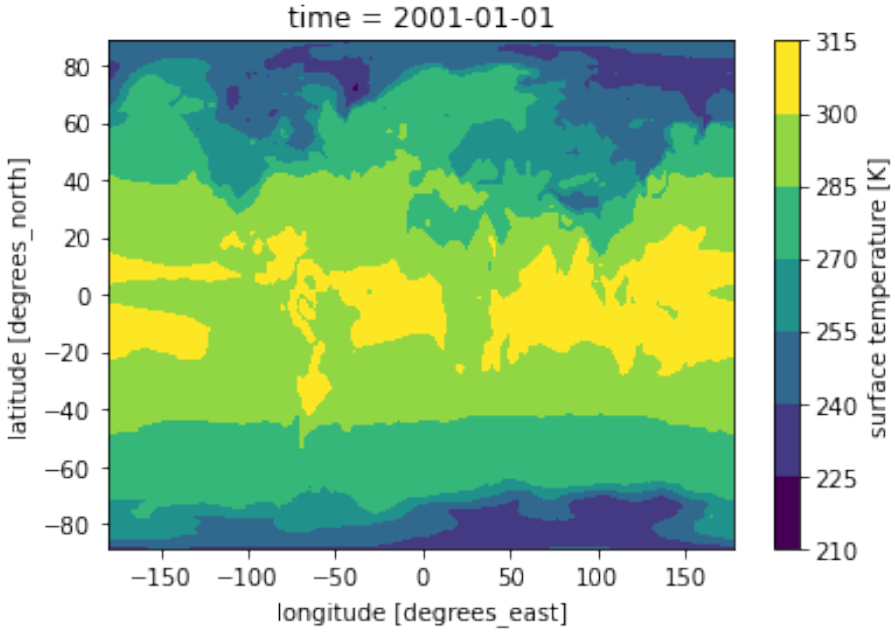
```
ds.tsurf[0,:,:].plot()
```

```
<matplotlib.collections.QuadMesh at 0x7fff6c4844c0>
```



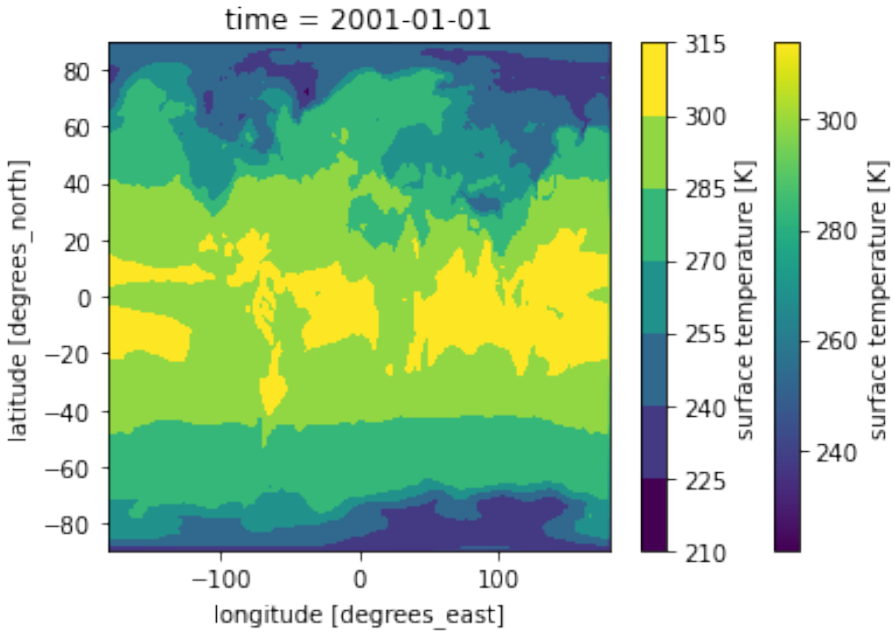

```
ds.tsurf[0,:,:].plot.contourf(levels=10)
```

```
<matplotlib.contour.QuadContourSet at 0x7fff6c69dd30>
```



```
ds.tsurf[0,:,:].plot()
ds.tsurf[0,:,:].plot.contourf(levels=10)
```

```
<matplotlib.contour.QuadContourSet at 0x7fff6cb17ac0>
```



More Plotting

Here comes another plot example to show the interpolation methods.

1. Create an Xarray DataArray using `xr.DataArray`, `np.linspace`and `np.sin`.
2. Plot array
3. Interpolate array values using *linear* and *cubic* methods and plot the interpolated data.

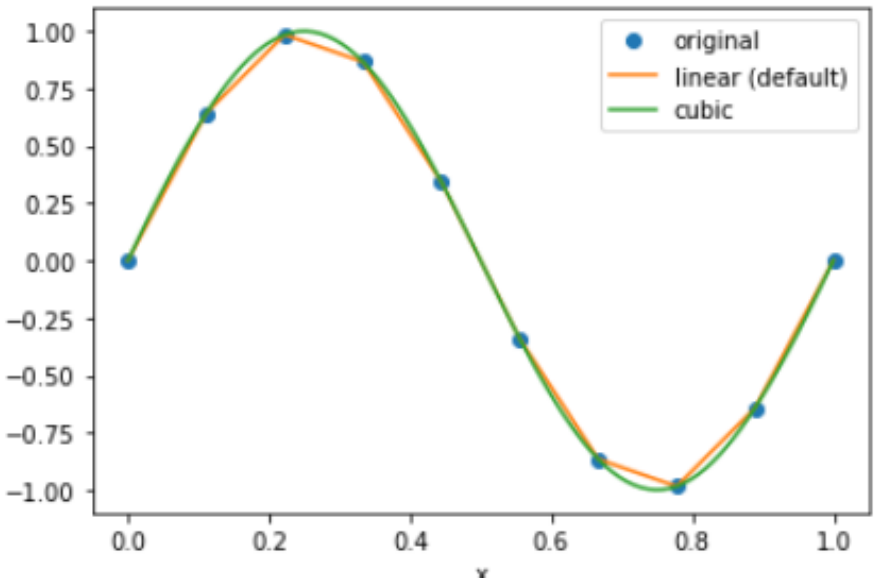
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(6,4))

da = xr.DataArray(np.sin(np.linspace(0, 2 * np.pi, 10)), dims="x", coords={"x": np.linspace(

da.plot.line('o', label='original')
da.interp(x=np.linspace(0, 1, 100)).plot.line(label='linear (default)')
```

```
da.interp(x=np.linspace(0, 1, 100), method='cubic').plot.line(label='cubic')
plt.legend()
```



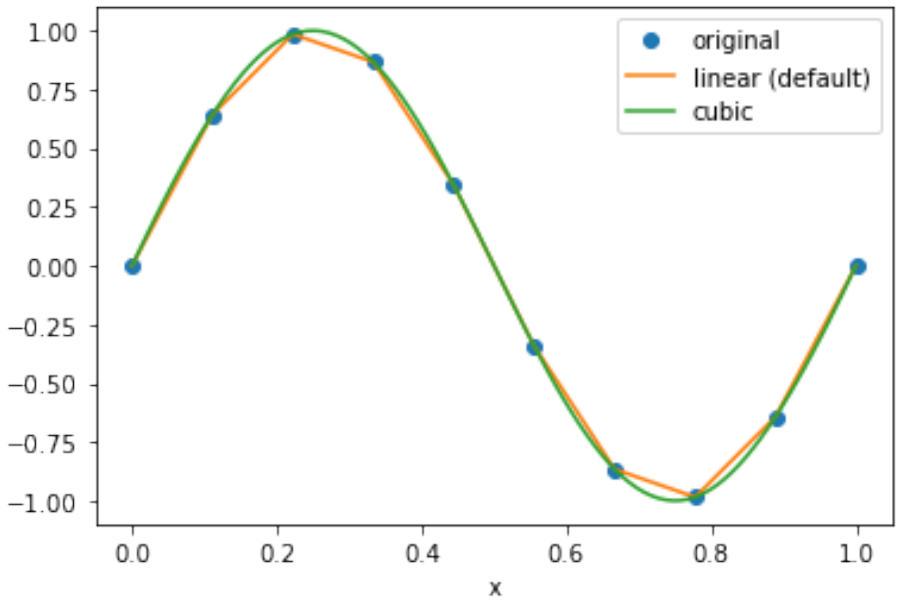

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(6,4))

da = xr.DataArray(np.sin(np.linspace(0, 2 * np.pi, 10)), dims="x", coords={"x": np.linspace(

da.plot.line('o', label='original')
da.interp(x=np.linspace(0, 1, 100)).plot.line(label='linear (default)')
da.interp(x=np.linspace(0, 1, 100), method='cubic').plot.line(label='cubic')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fff67a32e20>
```


original
linear (default)
cubic
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0
x