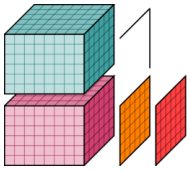
Xarray Introduction II



Xarray home page: https://xarray.pydata.org/en/stable/index.html
Xarray documentation: https://docs.xarray.dev/en/stable/index.html

How to create a Xarray DataSet with two 3D data variables

Importing required libraries

In [1]:
```
import xarray as xr
import numpy as np
import pandas as pd
import datetime
import matplotlib.pyplot as plt
import os, datetime
```

In [2]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Define your fake data

as lists for the coordinate variables of time, lat, lon

In [3]:
```
time = ['2023-01-01', '2023-01-02']
lat  = [45.,50.,55.,60.]
lon  = [0.,5.,10.,15.,20.]
```

create some random 3D numpy ndarray representing your dummy data

In [4]:
```
mydata1 = np.random.uniform(250,300,40).reshape((len(time),len(lat),len(lon)))
```

check the shape of your dummy data

In [5]:
```
mydata1.shape
```

Out[5]:
```
(2, 4, 5)
```

Create the first xarray DataArray (i.e. the first variable)

This is done by converting the ndarray dummy data plus the coordinate variables into an xarray DataArray.
Coordinates are added as dictionary in a key:value manner

In [6]:
```
da1 = xr.DataArray(data=mydata1,
                   coords={'time':  time,
                           'lat':   lat,
                           'lon':   lon,
                           },
                   attrs={'units': 'K', 'standard_name':'air_temperature'})
```

Check the DataArray by printing

In [7]:
```
da1
```

Out[7]: xarray.DataArray (**time**: 2, **lat**: 4, **lon**: 5)

array([[[262.58436778, 272.80238029, 298.77794124, 269.35762135,
274.18141368],
[266.64083243, 261.4119599 , 273.57778007, 298.46571515,
267.23154397],
[298.80524146, 263.330943 , 290.48984701, 279.28214477,
273.15722895],
[296.13833092, 284.71553328, 290.09587782, 278.66467638,
274.73057355]],

[[263.49924526, 272.32761298, 251.00406237, 257.17931112,
298.8519911],
[256.33394574, 250.03056502, 288.97120537, 270.17430925,
296.36966195],
[260.39699303, 274.79652963, 286.0245501 , 269.21030687,
256.43947251],
[255.8449326 , 266.92836174, 271.09775837, 290.90191851,
280.68422454]]])

▾ Coordinates:

time	(time)	<U10	'2023-01-01' '2023-01-02'
lat	(lat)	float64	45.0 50.0 55.0 60.0
lon	(lon)	float64	0.0 5.0 10.0 15.0 20.0

▸ Indexes: (3)

▾ Attributes:

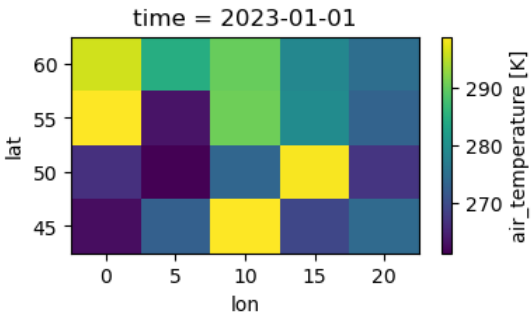
units :	K
standard_name :	air_temperature

check the first time step of the data by plotting

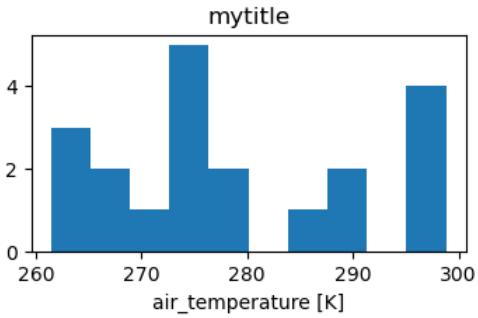
For 2D data, the default figure type of the plot() method is a pcolormesh.
In this case, the output of plot() and plot.pcolormesh() is identical. see
https://docs.xarray.dev/en/stable/generated/xarray.DataArray.plot.html

In [8]:
```
da1.isel(time=0).plot(aspect=2, size=2);

#-- the plot() method offers many configurations options of the plot, e.g. cmap="Blues"
#da1.isel(time=0,lat=slice(1,4)).plot(cmap="Blues")
```

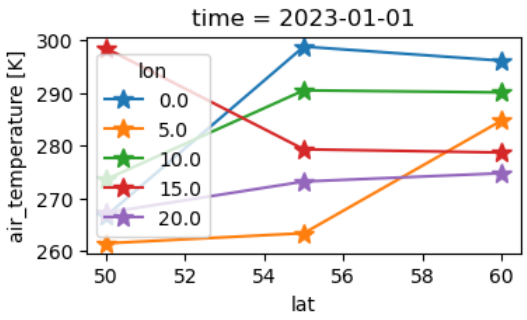


In [9]:
```
# you can also plot a histogram
da1.isel(time=0).plot.hist(aspect=2, size=2);
plt.title('mytitle');
```



you can also plot 2D data as a line plot with the plot.line() method, provided you specifiy one dimension as either x or y, which are then used as an axis of the line plot.

In [10]:
```
da1.isel(time=0,lat=slice(1,4)).plot.line(x="lat",marker='*',markersize=10, aspect=2, size=2);
```



Create the second xarray DataArray (i.e. the second variable)

In [11]:
```
#-- create a new DataArray by inheriting the attributes, dims, coords from da1
da2 = da1.copy(data=np.cos(mydata1)+10)
```

In [12]:
```
#-- overwrite the standard name inherited from da1
da2.attrs["standard_name"] = "canopy_temperature"
```

Convert the DataArrays into a DataSet

In [13]:
```
# it is mandatory to specify a name for each DataArray
ds = xr.Dataset({'airtemp': da1, 'cantemp': da2})
```

Look at the DataSet

In [14]:
```
ds
```

Out[14]: xarray.Dataset

▸ Dimensions: (**time**: 2, **lat**: 4, **lon**: 5)

▾ Coordinates:

time	(time)	<U10	'2023-01-01' '2023-01-02'
lat	(lat)	float64	45.0 50.0 55.0 60.0
lon	(lon)	float64	0.0 5.0 10.0 15.0 20.0

▾ Data variables:

airtemp	(time, lat, lon)	float64	262.6 272.8 298.8 ... 290.9 280.7
cantemp	(time, lat, lon)	float64	10.26 9.13 9.053 ... 9.7 9.531

▸ Indexes: (3)

▸ Attributes: (0)

Edit the global metadata

this is the metadata describing the entire dataset. For example, add the CF conventions, if you want to write out CF-compliant netCDF files in the end. In addition, it is advisable to add a title, a history, institution, references, etc. See also https://cfconventions.org/Data/cf-conventions/cf-conventions-1.10/cf-conventions.html section 2.6.2.

In [15]:
```
ds.attrs={"Conventions": "CF-1.10",
          "title": "this is just fake data for python course",
          "institution": "German Climate Computing Center (DKRZ)"}
```

In [16]:
```
ds
```

Out[16]: xarray.Dataset

▸ Dimensions: (**time**: 2, **lat**: 4, **lon**: 5)

▾ Coordinates:

time	(time)	<U10	'2023-01-01' '2023-01-02'
lat	(lat)	float64	45.0 50.0 55.0 60.0
lon	(lon)	float64	0.0 5.0 10.0 15.0 20.0

▾ Data variables:

airtemp	(time, lat, lon)	float64	262.6 272.8 298.8 ... 290.9 280.7
cantemp	(time, lat, lon)	float64	10.26 9.13 9.053 ... 9.7 9.531

▸ Indexes: (3)

▾ Attributes:

Conventions :	CF-1.10
title :	this is just fake data for python course
institution :	German Climate Computing Center (DKRZ)

In [17]:
```
#-- you can also edit the metadata afterwards
ds.lat.attrs={"standard_name": "latitude", "units": "degrees_north"}
```

Xarray functions/methods

`where` - to mask data

Similar to the NumPy `where` function, Xarray provides a `where` function that uses a condition to filter data. You can, e.g. filter the data by data value range or by a condition related to a dimension.

In [18]:
```
# For simplicity, we extract the data of interest from the Dataset prior the masking
```

In [19]:
```
var = ds["airtemp"].sel(time="2023-01-02")
```

Filter data by data range using `where` in a combined condition

In [20]:
```
mask1 = var.where((var > 273.15) & (var < 300.))
mask1
```

Out[20]: xarray.DataArray 'airtemp' (**lat**: 4, **lon**: 5)

array([[nan, nan, nan, nan, 298.8519911],
[nan, nan, 288.97120537, nan, 296.36966195],
[nan, 274.79652963, 286.0245501 , nan, nan],
[nan, nan, nan, 290.90191851, 280.68422454]])

▾ Coordinates:

time	()	<U10	'2023-01-02'
lat	(lat)	float64	45.0 50.0 55.0 60.0
lon	(lon)	float64	0.0 5.0 10.0 15.0 20.0

▸ Indexes: (2)

▾ Attributes:

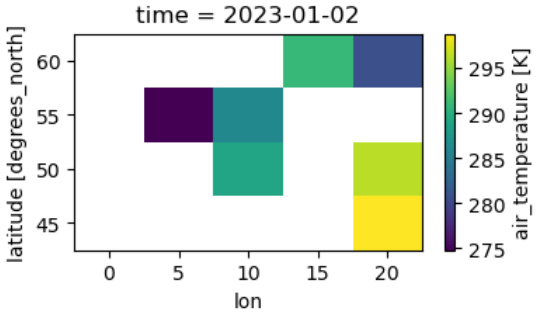
units :	K
standard_name :	air_temperature

In [21]:
```
mask1.dtype
```

Out[21]:
```
dtype('float64')
```

--> the masked data have the same shape as the original data
--> The masked data are of type float4 (mask1.dtype). The masked values are represented as nan.

In [22]:
```
mask1.plot(aspect=2, size=2);
```



The alternative masking to `where`

For using one mask as another mask, you need a bolean mask

In [23]:
```
mask2 = (var >= 280)
mask2
```

Out[23]: xarray.DataArray 'airtemp' (**lat**: 4, **lon**: 5)

array([[False, False, False, False, True],
[False, False, True, False, True],
[False, False, True, False, False],
[False, False, False, True, True]])

▼ Coordinates:

time	()	<U10	'2023-01-02'
lat	(lat)	float64	45.0 50.0 55.0 60.0
lon	(lon)	float64	0.0 5.0 10.0 15.0 20.0

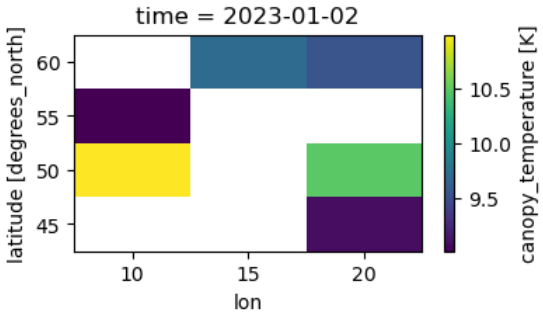
▶ Indexes: (2)

▶ Attributes: (0)

Example: Mask out ds['cantemp'] with mask2 and plot

In [24]:
```
ds['cantemp'].sel(time="2023-01-02").where(mask2, drop=True).plot(aspect=2, size=2)
```

Out[24]: <matplotlib.collections.QuadMesh at 0x7fff97951030>



Filter by a condition related to a dimension

In [25]:
```
vartmp= var.where(var.lon > 10.)
#vartmp.plot();
```

`isnull` - check where missing values exist

It returns a mask of True/False elements.

Our *var* variable does not contain missing values. We now define values below 273.15 as missing.

In [26]:
```
var = var.where(var < 273.15)
```

In [27]:
```
#-- to count the missings: count()
print(var.isnull().count().values)
```

```
20
```

`count` - count valid values

Count the data that are not missing values.

In [28]:
```
var.count().values    #-- with .values, the output will be printed as xarray DataArray
```

Out[28]:
```
array(13)
```

`notnull` - check where valid ("non-missing") values

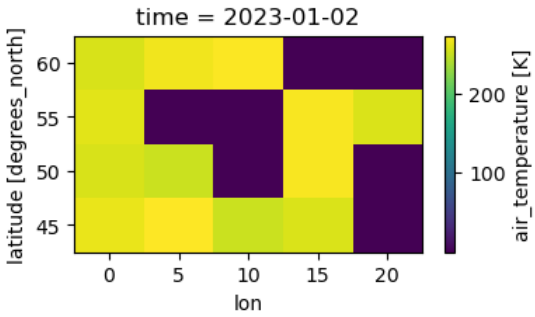
To check where valid ("non-missing") values are present, use `notnull`.

In [29]:
```
var.notnull().values
```

Out[29]:
```
array([[ True,  True,  True,  True, False],
       [ True,  True, False,  True, False],
       [ True, False, False,  True,  True],
       [ True,  True,  True, False, False]])
```

`fillna` - change missing value to a constant number

In [30]:
```
var.fillna(0.01).plot(aspect=2, size=2);
```



`min(), max(), mean(), sum(), std(), corr(), ...`

Xarray provides a lot of computational functions.

In [31]:
```
print(f'min = {ds.airtemp.min().values:6.2f},  max = {ds.airtemp.max().values:6.2f}')
```

```
min = 250.03,  max = 298.85
```

In [32]:
```
print(f'std = {ds.airtemp.std().values:6.2f}')
```

```
std =  14.20
```

Exercise 1

Use the precip data from the ../data/rectilinear_grid_2D.nc file:

1. compute the mean of the variable precip over 'time' and plot it
2. plot only the precip data > 0.0001
3. count how many non-missing values exist after masking values <=0.0001)
4. for the variable precip, compute the field mean, i.e. the mean over ('lat','lon')
5. plot the field mean as timeseries line plot and add a centered title

In [33]:
```
# 1.
```

In [34]:
```
# 2.
```

In [35]:
```
# 3.
```

In [36]:
```
# 4.
```

In [37]:
```
# 5.
```

Solution Exercise 1

In [38]:
```
# 1.
ds2 = xr.open_dataset('../data/rectilinear_grid_2D.nc')
ds2
```

Out[38]: 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 ... 176.2 178.1
lat	(lat)	float64	88.57 86.72 84.86 ... -86.72 -88.57

▾ Data variables:

tsurf	(time, lat, lon)	float32	...
precip	(time, lat, lon)	float32	...
u10	(time, lat, lon)	float32	...
v10	(time, lat, lon)	float32	...
slp	(time, lat, lon)	float32	...

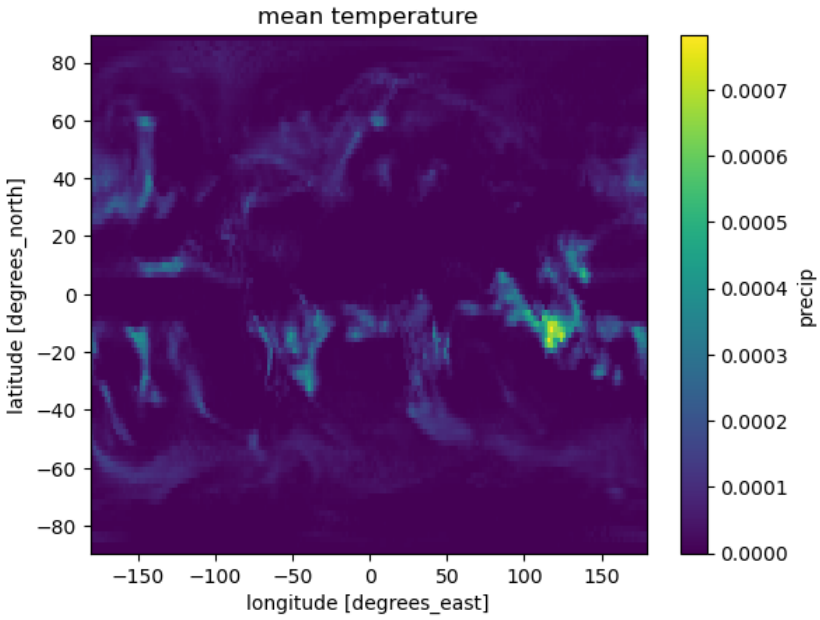
▸ Indexes: (3)

▾ Attributes:

CDI :	Climate Data Interface version 2.2.1 (https://mpimet.mpg.de/cdi)
Conventions :	CF-1.6
history :	Tue Oct 17 08:26:43 2023: cdo -seltimestep,1/12 -delname,qvi rectilinear_grid_2D.nc tmp.nc
CDO :	Climate Data Operators version 2.2.0 (https://mpimet.mpg.de/cdo)

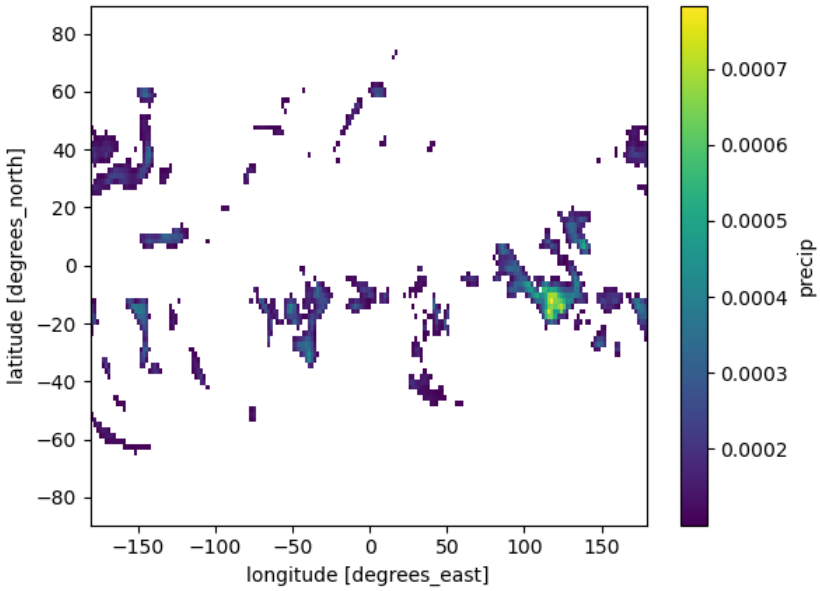
In [39]:
```
ds2m = ds2.precip.mean('time')
```

In [40]:
```
ds2m.plot();
plt.title("mean temperature");
```



In [41]:
```
# 2.
```

```
ds2m.where(ds2m > 0.0001).plot();     #-- to subset region: xlim=50, ylim=0
```



In [42]:
```
# 3.

print(ds2m.where(ds2m > 0.0001).count().values)
```

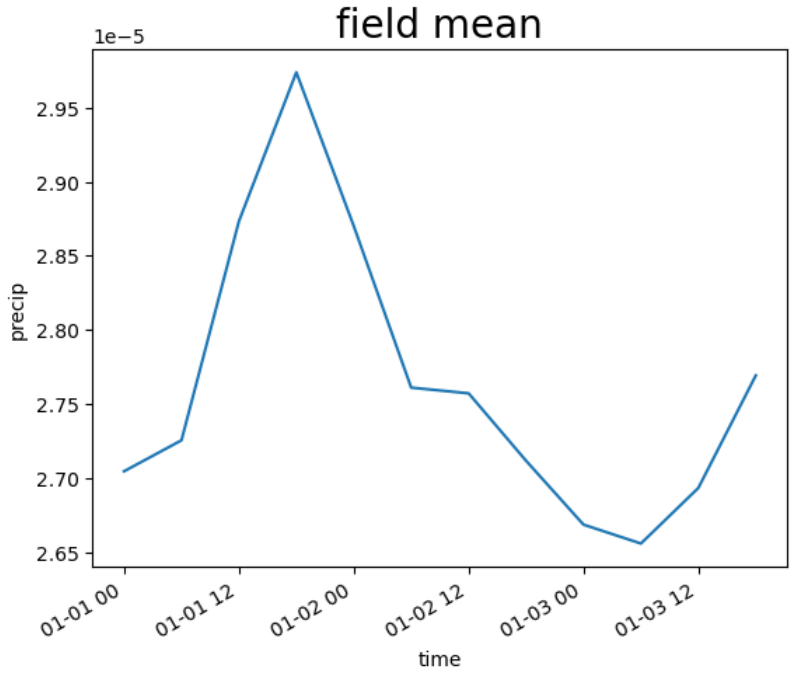
```
1278
```

In [43]:
```
# 4.

ds2fm = ds2.precip.mean(('lat','lon'))
```

In [44]:
```
# 5.

ax = plt.axes()
ds2fm.plot(ax=ax);
ax.set_title('field mean', loc='center', fontsize = 20);
```



Advanced Xarray Example: Climatology

For our next example, we use a historical and scenario global time series from CMIP6.

In [45]:
```
dir_data  = "../data/"
# historical data
fnameh = "hist_em_LR_temp_subset_1980-2000.nc"
# scenario data
fname  = "ssp245_em_LR_temp_subset_2070-2100.nc"
```

Create historical climatology map

Extract some time period

First, we extract a 20-year time range (1980-2000).

In [46]:
```
dsh = xr.open_dataset(dir_data + fnameh)
dsh
```

Out[46]: xarray.Dataset

▸ Dimensions: (**time**: 372, bnds: 2, **lon**: 192, **lat**: 96)

▾ Coordinates:

time	(time)	datetime64[ns]	1970-01-16T12:00:00 ... 2000-12-...	
lon	(lon)	float64	-180.0 -178.1 ... 176.2 178.1	
lat	(lat)	float64	-88.57 -86.72 ... 86.72 88.57	
height	()	float64	...	

▾ Data variables:

time_bnds	(time, bnds)	datetime64[ns]	...	
lon_bnds	(lon, bnds)	float64	...	
lat_bnds	(lat, bnds)	float64	...	
tas	(time, lat, lon)	float32	...	

▸ Indexes: (3)

▸ Attributes: (50)

Compute monthly means

Xarray allows us to group the time steps monthly-wise with the `groupby` method and compute the monthly means.

In [47]:
```
clim_xr = dsh.groupby('time.month').mean()   #-- corresponding to cdo ymonmean
```

In [48]:
```
clim_xr
```

Out[48]: xarray.Dataset

▸ Dimensions: (**lon**: 192, **month**: 12, bnds: 2, **lat**: 96)

▾ Coordinates:

lon	(lon)	float64	-180.0 -178.1 ... 176.2 178.1	
lat	(lat)	float64	-88.57 -86.72 ... 86.72 88.57	
height	()	float64	2.0	
month	(month)	int64	1 2 3 4 5 6 7 8 9 10 11 12	

▾ Data variables:

lon_bnds	(month, lon, bnds)	float64	179.1 -179.1 180.9 ... 537.2 179.1	
lat_bnds	(month, lat, bnds)	float64	-89.5 -87.65 -87.65 ... 87.65 89.5	
tas	(month, lat, lon)	float32	243.2 243.2 243.1 ... 246.0 246.0	

▸ Indexes: (3)

▸ Attributes: (50)

Compute the area weights

To account for the different grid cell sizes, we calculate the variable weighted this time.

In [49]:
```
weights = np.cos(np.deg2rad(dsh.lat))
```

In [50]:
```
weights
```

Out[50]: xarray.DataArray 'lat' (**lat**: 96)

array([0.02491778, 0.05717144, 0.08955539, 0.12188768, 0.15410725,
0.1861715 , 0.21804295, 0.24968611, 0.28106651, 0.31215034,
0.34290431, 0.37329559, 0.40329182, 0.43286109, 0.46197199,
0.49059359, 0.51869553, 0.54624798, 0.57322171, 0.59958812,
0.62531924, 0.65038779, 0.6747672 , 0.6984316 , 0.72135592,
0.74351585, 0.76488791, 0.78544943, 0.80517862, 0.82405458,
0.84205728, 0.85916766, 0.87536757, 0.89063985, 0.9049683 ,
0.91833775, 0.93073401, 0.94214396, 0.9525555 , 0.96195759,
0.97034027, 0.97769466, 0.98401296, 0.98928847, 0.9935156 ,
0.99668988, 0.99880794, 0.99986753, 0.99986753, 0.99880794,
0.99668988, 0.9935156 , 0.98928847, 0.98401296, 0.97769466,
0.97034027, 0.96195759, 0.9525555 , 0.94214396, 0.93073401,
0.91833775, 0.9049683 , 0.89063985, 0.87536757, 0.85916766,
0.84205728, 0.82405458, 0.80517862, 0.78544943, 0.76488791,
0.74351585, 0.72135592, 0.6984316 , 0.6747672 , 0.65038779,
0.62531924, 0.59958812, 0.57322171, 0.54624798, 0.51869553,
0.49059359, 0.46197199, 0.43286109, 0.40329182, 0.37329559,
0.34290431, 0.31215034, 0.28106651, 0.24968611, 0.21804295,
0.1861715 , 0.15410725, 0.12188768, 0.08955539, 0.05717144,
0.02491778])

▼ Coordinates:

lat	(lat)	float64	-88.57 -86.72 ... 86.72 88.57
height	()	float64	...

▶ Indexes: (1)

▼ Attributes:

standard_name :	latitude
long_name :	Latitude
units :	degrees_north
axis :	Y
bounds :	lat_bnds

Compute the weighted data.

In [51]:
```
clim_xr_wgt = clim_xr.weighted(weights)
```

In [52]:
```
clim_xr_wgt
```

Out[52]:
```
DatasetWeighted with weights along dimensions: lat
```

In [53]:
```
#-- The weighted call returns a class xarray.core.weighted.DatasetWeighted(obj, weights).
#-- To obtain the weights as xarray DataArray, use clim_xr_wgt.weights
clim_xr_wgt.weights.head(5).values    #-- head(5) shows the first 5 values, only.
```

Out[53]:
```
array([0.02491778, 0.05717144, 0.08955539, 0.12188768, 0.15410725])
```

Compute the spatial mean

In [54]:
```
clim_xr_mean = clim_xr_wgt.mean(('lat','lon'))
```

In [55]:
```
clim_xr_mean
```

Out[55]: xarray.Dataset

▶ Dimensions: (**month**: 12, bnds: 2)

▼ Coordinates:

height	()	float64	2.0
month	(month)	int64	1 2 3 4 5 6 7 8 9 10 11 12

▼ Data variables:

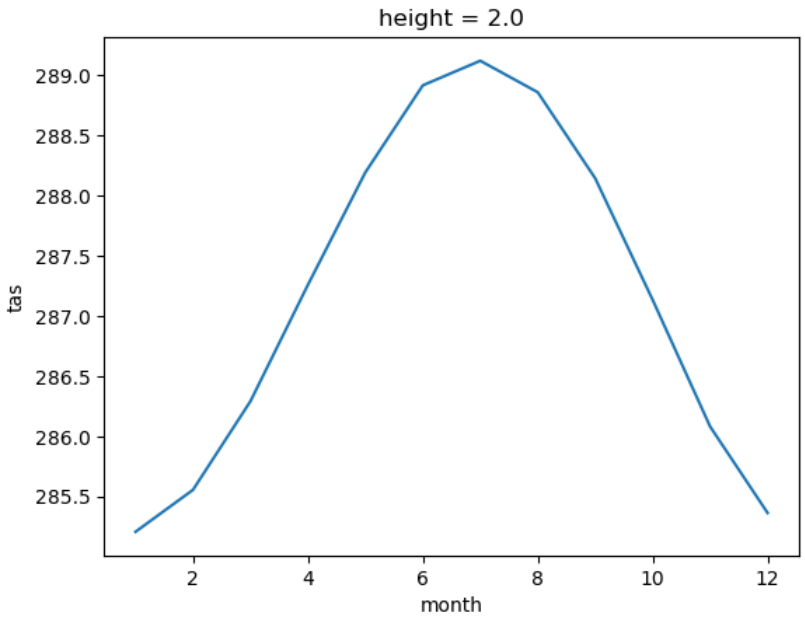
lon_bnds	(month, bnds)	float64	358.1 -2.776e-15 ... -2.776e-15
lat_bnds	(month, bnds)	float64	-0.9326 0.9326 ... -0.9326 0.9326
tas	(month)	float64	285.2 285.6 286.3 ... 286.1 285.4

▶ Indexes: (1)

▶ Attributes: (0)

Plot the result.

In [56]:
```
clim_xr_mean.tas.plot();
```



Derive anomaly from the scenario and the historical climatology

Open the scenario file and extract the time range 2080-2100.

In [57]:
```
ds = xr.open_dataset(dir_data + fname)
ds = ds.sel(time=slice('2080-01-01','2100-12-31'))
```

Compute the spatial mean of the weighted data.

In [58]:
```
tas_xr_wgt = ds.tas.weighted(weights).mean(('lat','lon'))
```

Compute the anomaly. This is done by subtracting the monthly climatology from the monthly grouped data.

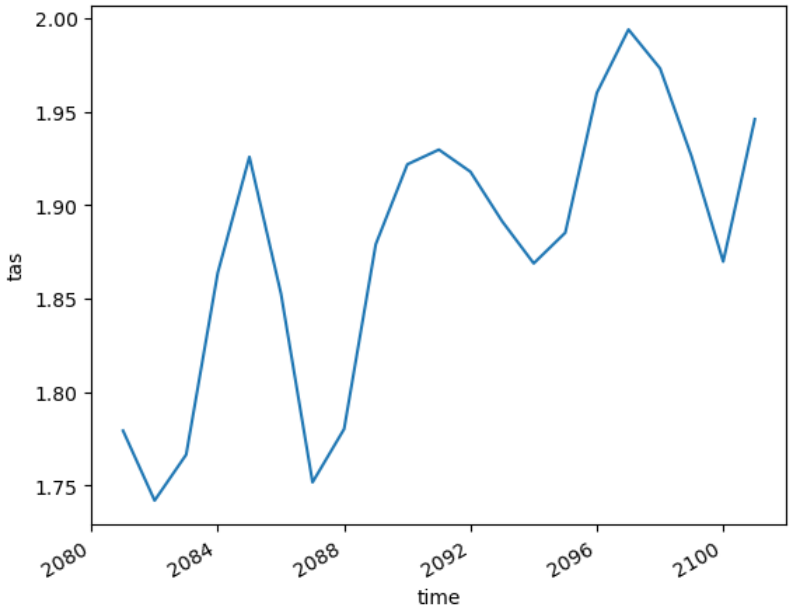
In [59]:
```
anom_xr = tas_xr_wgt.groupby('time.month') - clim_xr_mean
```

Compute the yearly means.

In [60]:
```
anom_xr_ymean = anom_xr.resample(time='Y').mean()
```

Plot the result.

In [61]:
```
anom_xr_ymean.tas.plot();
```



See also:

- Project Pythia Computations and Masks with Xarray https://foundations.projectpythia.org/core/xarray/computation-masking.html
- Tutorials and Videos https://docs.xarray.dev/en/stable/tutorials-and-videos.html
- DKRZ tutorials https://data-infrastructure-services.gitlab-pages.dkrz.de/tutorials-and-use-cases/Tutorials.html
- Pangeo Xarray Tutorial http://gallery.pangeo.io/repos/pangeo-data/pangeo-tutorial-gallery/xarray.html
- Copernicus https://ecmwf-projects.github.io/copernicus-training-c3s/reanalysis-climatology.html#anomaly-calculation