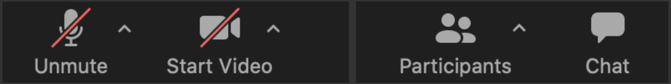
Working with multidimensional datasets in xarray

Alex Razoumov
alex.razoumov@westgrid.ca

Zoom controls

- Please mute your microphone and camera unless you have a question
- To ask questions at any time, type in Chat, or Unmute to ask via audio
- Raise your hand in Participants



- This talk is being recorded
  - your name might appear in the recording (if you want, you can change it)
  - video will be posted at `https://westgrid.github.io/trainingMaterials` under one of the topics
- Email training@westgrid.ca

Starting point

I assume basic familiarity with numpy arrays

As a quick recap (next few slides), numpy:

- provides a mechanism for uniform collections (aka arrays) of fixed-size, fixed-type items with contiguous allocation in memory
  - $\Rightarrow$ large performance benefits (no reading of extra bits, no type checking, all compiled code)

Starting point

I assume basic familiarity with numpy arrays

As a quick recap (next few slides), numpy:

- provides a mechanism for uniform collections (aka arrays) of fixed-size, fixed-type items with contiguous allocation in memory
  - ⇒ large performance benefits (no reading of extra bits, no type checking, all compiled code)
- arrays behave very differently from Python lists
- implements universal (vectorized) functions on a large number of elements
  - an operation on the array is applied to each element
  - statically typed, compiled routine
  - ⇒ almost always much faster than native Python code, especially for large and/or multidimensional arrays
  - ⇒ use these whenever possible to replace native `for` loops

Starting point

I assume basic familiarity with numpy arrays

As a quick recap (next few slides), numpy:

- provides a mechanism for uniform collections (aka arrays) of fixed-size, fixed-type items with contiguous allocation in memory
  - ⇒ large performance benefits (no reading of extra bits, no type checking, all compiled code)
- arrays behave very differently from Python lists
- implements universal (vectorized) functions on a large number of elements
  - an operation on the array is applied to each element
  - statically typed, compiled routine
  - ⇒ almost always much faster than native Python code, especially for large and/or multidimensional arrays
  - ⇒ use these whenever possible to replace native `for` loops
- provides the ability to operate between arrays of different sizes and shapes ("broadcasting")

Starting point

I assume basic familiarity with numpy arrays

As a quick recap (next few slides), numpy:

- provides a mechanism for uniform collections (aka arrays) of fixed-size, fixed-type items with contiguous allocation in memory
  - ⇒ large performance benefits (no reading of extra bits, no type checking, all compiled code)
- arrays behave very differently from Python lists
- implements universal (vectorized) functions on a large number of elements
  - an operation on the array is applied to each element
  - statically typed, compiled routine
  - ⇒ almost always much faster than native Python code, especially for large and/or multidimensional arrays
  - ⇒ use these whenever possible to replace native `for` loops
- provides the ability to operate between arrays of different sizes and shapes ("broadcasting")
- provides linear algebra operations on mathematical arrays including linear solve and various decompositions

Vectorized functions

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a**2          # each element is a square of the corresponding element of a
array([0, 1, 4, 9, 16, 25, 36, 49, 64, 81])
```

Vectorized functions

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a**2          # each element is a square of the corresponding element of a
array([0, 1, 4, 9, 16, 25, 36, 49, 64, 81])

>>> np.log10(a+1)
array([0., 0.30103, 0.47712125, 0.60205999, 0.69897, 0.77815125, 0.84509804, 0.90308999, 0.95424251, 1.])
```

Vectorized functions

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a**2          # each element is a square of the corresponding element of a
array([0, 1, 4, 9, 16, 25, 36, 49, 64, 81])

>>> np.log10(a+1)
array([0., 0.30103, 0.47712125, 0.60205999, 0.69897, 0.77815125, 0.84509804, 0.90308999, 0.95424251, 1.])

>>> (a**2+a)/(a+1)    # the result should effectively be a floating-version copy of a
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

Vectorized functions

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a**2          # each element is a square of the corresponding element of a
array([0, 1, 4, 9, 16, 25, 36, 49, 64, 81])

>>> np.log10(a+1)
array([0., 0.30103, 0.47712125, 0.60205999, 0.69897, 0.77815125, 0.84509804, 0.90308999, 0.95424251, 1.])

>>> (a**2+a)/(a+1)     # the result should effectively be a floating-version copy of a
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])

>>> np.arange(10) / np.arange(1,11)  # this is np.array([ 0/1, 1/2, 2/3, 3/4, ..., 9/10 ])
array([0., 0.5, 0.66666667, 0.75, 0.8, 0.83333333, 0.85714286, 0.875, 0.88888889, 0.9])
```

Replace `for` loops with vectorized functions

Let's check this summation

$$\sum_{k=1}^{\infty} \frac{k^2}{2^k} = 6$$

Replace `for` loops with vectorized functions

Let's check this summation

$$\sum_{k=1}^{\infty} \frac{k^2}{2^k} = 6$$

```
>>> k = np.arange(1,11)    # let's try the first 10 terms
>>> sum(k**2/2**k)
5.857421875
```

Replace `for` loops with vectorized functions

Let's check this summation

$$\sum_{k=1}^{\infty} \frac{k^2}{2^k} = 6$$

```
>>> k = np.arange(1,11)    # let's try the first 10 terms
>>> sum(k**2/2**k)
5.857421875

>>> k = np.arange(1,51)    # the first 50 terms
>>> sum(k**2/2**k)
5.999999999997597
```

Replace `for` loops with vectorized functions

Let's check this summation

$$\sum_{k=1}^{\infty} \frac{k^2}{2^k} = 6$$

```
>>> k = np.arange(1,11)    # let's try the first 10 terms
>>> sum(k**2/2**k)
5.857421875

>>> k = np.arange(1,51)    # the first 50 terms
>>> sum(k**2/2**k)
5.999999999997597

>>> k = np.arange(1,101)    # the first 100 terms
>>> sum(k**2/2**k)
<stdin>:1: RuntimeWarning: divide by zero encountered in true_divide
inf
>>> sum(k**2/2.**k)
5.999999999999999
```

Broadcasting rules

1. The shape of an array with fewer dimensions is padded with 1's on the left
2. Any array with shape equal to 1 in that dimension is stretched to match the other array's shape
3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised

Broadcasting rules

1. The shape of an array with fewer dimensions is padded with 1's on the left
2. Any array with shape equal to 1 in that dimension is stretched to match the other array's shape
3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised

Example 1:

a: (3,) $\rightarrow$ (1,3) $\rightarrow$ (3,3)
b: (3,1) $\rightarrow$ (3,1) $\rightarrow$ (3,3)
a+b: $\rightarrow$ (3,3)

Broadcasting rules

1. The shape of an array with fewer dimensions is padded with 1's on the left
2. Any array with shape equal to 1 in that dimension is stretched to match the other array's shape
3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised

Example 1:

a: (3,) $\rightarrow$ (1,3) $\rightarrow$ (3,3)
b: (3,1) $\rightarrow$ (3,1) $\rightarrow$ (3,3)
a+b: $\rightarrow$ (3,3)

Example 2:

a: (3,) $\rightarrow$ (1,3) $\rightarrow$ (3,3)
b: (3,2) $\rightarrow$ (3,2) $\rightarrow$ (3,2)
a+b: $\rightarrow$ error

Broadcasting rules

1. The shape of an array with fewer dimensions is padded with 1's on the left
2. Any array with shape equal to 1 in that dimension is stretched to match the other array's shape
3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised

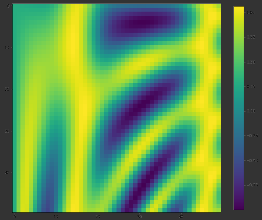
Example 1:

a: (3,) $\rightarrow$ (1,3) $\rightarrow$ (3,3)
b: (3,1) $\rightarrow$ (3,1) $\rightarrow$ (3,3)
a+b: $\rightarrow$ (3,3)

Example 2:

a: (3,) $\rightarrow$ (1,3) $\rightarrow$ (3,3)
b: (3,2) $\rightarrow$ (3,2) $\rightarrow$ (3,2)
a+b: $\rightarrow$ error

```python
import numpy as np
import matplotlib.pyplot as plt
plt.figure(figsize=(12,12))
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50).reshape(50,1)
z = np.sin(x)**8 + np.cos(5+x*y)*np.cos(x)     # broadcast!
plt.imshow(z)
plt.colorbar(shrink=0.8)
```



Many other Python packages were built on top of numpy

- scikit-image is a collection of algorithms for image processing
  - each image is stored as width $\times$ height $\times$ 4 (R,G,B,α channels) numpy array
- pandas for (2D) tabular data manipulation
- yt for volumetric multi-resolution data
  - multiple subgrids at different resolution levels
  - any 3D field on each subgrid is stored as a 3D numpy array
  - any 3D field across all subgrids can be accessed as a flattened array
  - dozens of convenient functions to manipulate and visualize yt datasets
  - can import multi-resolution data from 20+ different file formats
  - fairly easy to convert generic array data into yt format
  - covered in our two earlier webinars:
    - *"Using YT for analysis and visualization of volumetric data"* (November 2018)
    - *"Working with data objects in YT"* (January 2019)
    - recordings at `https://bit.ly/vispages`

Xarray library

- ✔ Built on top of numpy and pandas
- ✔ Brings the power of pandas (easy data manipulation, I/O, plotting) to multidimensional arrays
- ✘ Does not support multi-resolution data out-of-the-box
  - i.e. you cannot define a single 3D field that spans multiple grids
  - however, you *can* define multiple arrays on top of multiple grids inside the same dataset
- ✔ Not limited to 3D $\Rightarrow$ data of any dimensionality
- ✔ Makes it easy to work with time-dependent arrays (time = one of the dimensions)

- Two main data structures in xarray:
  - *xarray.DataArray* is a fancy, labelled version of *numpy.ndarray*
  - *xarray.Dataset* is a collection of multiple *xarray.DataArray*'s that (usually) share dimensions

DataArray

```
import xarray as xr
import numpy as np
data = xr.DataArray(
    np.random.random(size=(4,3)),
    dims=("y","x"),          # we want 'y' to represent rows and 'x' columns
    coords={"x": [10,11,12],
            "y": [10,20,30,40]}  # coordinate labels/values
)
print(data)
```

DataArray

```
import xarray as xr
import numpy as np
data = xr.DataArray(
    np.random.random(size=(4,3)),
    dims=("y","x"),          # we want 'y' to represent rows and 'x' columns
    coords={"x": [10,11,12],
            "y": [10,20,30,40]}  # coordinate labels/values
)
print(data)
```

```
<xarray.DataArray (y: 4, x: 3)>
array([[0.8468579 , 0.79008336, 0.67300866],
       [0.61169687, 0.82379812, 0.05382901],
       [0.70889338, 0.01498123, 0.50806224],
       [0.3437665 , 0.84299556, 0.89612048]])
Coordinates:
  * x        (x) int64 10 11 12
  * y        (y) int64 10 20 30 40
```

Coordinates and attributes

```
>>> data.dims
('y', 'x')
>>> data.size, data.dtype
(12, dtype('float64'))
>>> data.coords
Coordinates:
  * x        (x) int64 10 11 12
  * y        (y) int64 10 20 30 40
```

Coordinates and attributes

```
>>> data.dims
('y', 'x')
>>> data.size, data.dtype
(12, dtype('float64'))
>>> data.coords
Coordinates:
  * x        (x) int64 10 11 12
  * y        (y) int64 10 20 30 40
```

access specific coordinate:

```
>>> data.coords['x']
<xarray.DataArray 'x' (x: 3)>
array([10, 11, 12])
Coordinates:
  * x        (x) int64 10 11 12
```

Coordinates and attributes

```
>>> data.dims
('y', 'x')
>>> data.size, data.dtype
(12, dtype('float64'))
>>> data.coords
Coordinates:
  * x        (x) int64 10 11 12
  * y        (y) int64 10 20 30 40
```

access specific coordinate:

```
>>> data.coords['x']
<xarray.DataArray 'x' (x: 3)>
array([10, 11, 12])
Coordinates:
  * x        (x) int64 10 11 12

>>> data.coords['x'][1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x       int64 11
```

Coordinates and attributes

```
>>> data.dims
('y', 'x')
>>> data.size, data.dtype
(12, dtype('float64'))
>>> data.coords
Coordinates:
  * x          (x) int64 10 11 12
  * y          (y) int64 10 20 30 40
```

access specific coordinate:

```
>>> data.coords['x']
<xarray.DataArray 'x' (x: 3)>
array([10, 11, 12])
Coordinates:
  * x          (x) int64 10 11 12

>>> data.coords['x'][1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x          int64 11
```

can also use pandas-like notation:

```
>>> data.x[1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x          int64 11
```

Coordinates and attributes

```
>>> data.dims
('y', 'x')
>>> data.size, data.dtype
(12, dtype('float64'))
>>> data.coords
Coordinates:
  * x         (x) int64 10 11 12
  * y         (y) int64 10 20 30 40
```

access specific coordinate:

```
>>> data.coords['x']
<xarray.DataArray 'x' (x: 3)>
array([10, 11, 12])
Coordinates:
  * x         (x) int64 10 11 12

>>> data.coords['x'][1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x         int64 11
```

can also use pandas-like notation:

```
>>> data.x[1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x         int64 11

>>> data.x.values
array([10, 11, 12])
```

Coordinates and attributes

```
>>> data.dims
('y', 'x')
>>> data.size, data.dtype
(12, dtype('float64'))
>>> data.coords
Coordinates:
  * x         (x) int64 10 11 12
  * y         (y) int64 10 20 30 40
```

access specific coordinate:

```
>>> data.coords['x']
<xarray.DataArray 'x' (x: 3)>
array([10, 11, 12])
Coordinates:
  * x         (x) int64 10 11 12

>>> data.coords['x'][1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x         int64 11
```

can also use pandas-like notation:

```
>>> data.x[1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x         int64 11

>>> data.x.values
array([10, 11, 12])
```

```
>>> data.attrs = {"author": "AR", "date": "2020-09-30"}
>>> data.attrs["name"] = "density"
>>> data.attrs["units"] = "g/cm^3"
>>> data.x.attrs["units"] = "cm"
>>> data.y.attrs["units"] = "cm"
```

Coordinates and attributes

```
>>> data.dims
('y', 'x')
>>> data.size, data.dtype
(12, dtype('float64'))
>>> data.coords
Coordinates:
  * x          (x) int64 10 11 12
  * y          (y) int64 10 20 30 40
```

access specific coordinate:

```
>>> data.coords['x']
<xarray.DataArray 'x' (x: 3)>
array([10, 11, 12])
Coordinates:
  * x          (x) int64 10 11 12

>>> data.coords['x'][1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x          int64 11
```

can also use pandas-like notation:

```
>>> data.x[1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x          int64 11

>>> data.x.values
array([10, 11, 12])
```

```
>>> data.attrs = {"author": "AR", "date": "2020-09-30"}
>>> data.attrs["name"] = "density"
>>> data.attrs["units"] = "g/cm^3"
>>> data.x.attrs["units"] = "cm"
>>> data.y.attrs["units"] = "cm"


>>> data.attrs     # all top-level attributes
{'author':'AR', 'date':'2020-09-30', 'name':'density', 'units':'g/cm^3'}
```

Coordinates and attributes

```
>>> data.dims
('y', 'x')
>>> data.size, data.dtype
(12, dtype('float64'))
>>> data.coords
Coordinates:
  * x        (x) int64 10 11 12
  * y        (y) int64 10 20 30 40
```

access specific coordinate:

```
>>> data.coords['x']
<xarray.DataArray 'x' (x: 3)>
array([10, 11, 12])
Coordinates:
  * x        (x) int64 10 11 12

>>> data.coords['x'][1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x        int64 11
```

can also use pandas-like notation:

```
>>> data.x[1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x        int64 11

>>> data.x.values
array([10, 11, 12])
```

```
>>> data.attrs = {"author": "AR", "date": "2020-09-30"}
>>> data.attrs["name"] = "density"
>>> data.attrs["units"] = "g/cm^3"
>>> data.x.attrs["units"] = "cm"
>>> data.y.attrs["units"] = "cm"


>>> data.attrs    # all top-level attributes
{'author':'AR', 'date':'2020-09-30', 'name':'density', 'units':'g/cm^3'}


>>> data          # all top-level attributes show here as well
<xarray.DataArray (y: 4, x: 3)>
array([[0.8468579 , 0.79008336, 0.999999  ],
       [0.61169687, 0.82379812, 0.05382901],
       [0.70889338, 0.01498123, 0.50806224],
       [0.3437665 , 0.84299556, 0.89612048]])
Coordinates:
  * x        (x) int64 10 11 12
  * y        (y) int64 10 20 30 40
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3
```

Coordinates and attributes

```
>>> data.dims
('y', 'x')
>>> data.size, data.dtype
(12, dtype('float64'))
>>> data.coords
Coordinates:
  * x         (x) int64 10 11 12
  * y         (y) int64 10 20 30 40
```

access specific coordinate:

```
>>> data.coords['x']
<xarray.DataArray 'x' (x: 3)>
array([10, 11, 12])
Coordinates:
  * x         (x) int64 10 11 12

>>> data.coords['x'][1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x         int64 11
```

can also use pandas-like notation:

```
>>> data.x[1]
<xarray.DataArray 'x' ()>
array(11)
Coordinates:
    x         int64 11

>>> data.x.values
array([10, 11, 12])
```

```
>>> data.attrs = {"author": "AR", "date": "2020-09-30"}
>>> data.attrs["name"] = "density"
>>> data.attrs["units"] = "g/cm^3"
>>> data.x.attrs["units"] = "cm"
>>> data.y.attrs["units"] = "cm"


>>> data.attrs    # all top-level attributes
{'author':'AR', 'date':'2020-09-30', 'name':'density', 'units':'g/cm^3'}


>>> data          # all top-level attributes show here as well
<xarray.DataArray (y: 4, x: 3)>
array([[0.8468579 , 0.79008336, 0.999999  ],
       [0.61169687, 0.82379812, 0.05382901],
       [0.70889338, 0.01498123, 0.50806224],
       [0.3437665 , 0.84299556, 0.89612048]])
Coordinates:
  * x         (x) int64 10 11 12
  * y         (y) int64 10 20 30 40
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3


>>> data.x        # only 'x' attributes
<xarray.DataArray 'x' (x: 3)>
array([10, 11, 12])
Coordinates:
  * x         (x) int64 10 11 12
Attributes:
    units:    cm
```

Subsetting arrays

- Using the usual Python square brackets

```
data[0,:]                  # first row, also a DataArray
data[:,-2:]                # last two columns
data[-1,-1] = 0.999999     # can modify in-place
data.values[0,0:]          # first row, numpy array
```

- `DataArray.isel()` selects by coordinate index (single index, list, range)
- `DataArray.sel()` selects by coordinate value (single value, list, range)
- `DataArray.interp()` interpolates by coordinate value

DataArray.isel() to select by coordinate index

- `data.isel()` is the same as `data` (no selection)
- Can refine selection by passing any number of arguments

DataArray.isel() to select by coordinate index

- `data.isel()` is the same as `data` (no selection)
- Can refine selection by passing any number of arguments

```
>>> data.isel(y=1)       # second row
<xarray.DataArray (x: 3)>
array([0.54756005, 0.21070062, 0.71370984])
Coordinates:
  * x        (x) int64 10 11 12
    y        int64 20
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3

>>> type(data.isel(y=1))
<class 'xarray.core.dataarray.DataArray'>
```

DataArray.isel() to select by coordinate index

- `data.isel()` is the same as `data` (no selection)
- Can refine selection by passing any number of arguments

```
>>> data.isel(y=1)        # second row
<xarray.DataArray (x: 3)>
array([0.54756005, 0.21070062, 0.71370984])
Coordinates:
  * x        (x) int64 10 11 12
    y        int64 20
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3

>>> type(data.isel(y=1))
<class 'xarray.core.dataarray.DataArray'>
```

```
>>> data.isel(y=0, x=[-2,-1])
           # first row, last two columns
<xarray.DataArray (x: 2)>
array([0.73093109, 0.85287453])
Coordinates:
  * x        (x) int64 11 12
    y        int64 10
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
```

DataArray.isel() to select by coordinate index

- `data.isel()` is the same as `data` (no selection)
- Can refine selection by passing any number of arguments

```
>>> data.isel(y=1)        # second row
<xarray.DataArray (x: 3)>
array([0.54756005, 0.21070062, 0.71370984])
Coordinates:
  * x        (x) int64 10 11 12
    y        int64 20
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3

>>> type(data.isel(y=1))
<class 'xarray.core.dataarray.DataArray'>
```

```
>>> data.isel(y=0, x=[-2,-1])
           # first row, last two columns
<xarray.DataArray (x: 2)>
array([0.73093109, 0.85287453])
Coordinates:
  * x        (x) int64 11 12
    y        int64 10
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density

>>> data.isel(y=0, x=[-2,-1]).values
```

DataArray.sel() to select by coordinate value

```
>>> data.x.dtype
dtype('int64')
>>> data.x.values
array([10, 11, 12])
```

DataArray.sel() to select by coordinate value

```
>>> data.x.dtype
dtype('int64')
>>> data.x.values
array([10, 11, 12])

>>> data.sel(x=10)
<xarray.DataArray (y: 4)>
array([0.4515217, 0.54756005,
       0.506477, 0.73067568])
Coordinates:
    x        int64 10
  * y        (y) int64 10 20 30 40
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3
```

DataArray.sel() to select by coordinate value

```
>>> data.x.dtype
dtype('int64')
>>> data.x.values
array([10, 11, 12])

>>> data.sel(x=10)
<xarray.DataArray (y: 4)>
array([0.4515217, 0.54756005,
       0.506477, 0.73067568])
Coordinates:
    x        int64 10
  * y        (y) int64 10 20 30 40
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3
```

```
>>> data.y
<xarray.DataArray 'y' (y: 4)>
array([10, 20, 30, 40])
Coordinates:
  * y          (y) int64 10 20 30 40
Attributes:
    units:    cm

>>> data.sel(x=10,y=[30,40]).values
array([0.506477  , 0.73067568])
```

DataArray.sel() to select by coordinate value

```
>>> data.x.dtype
dtype('int64')
>>> data.x.values
array([10, 11, 12])

>>> data.sel(x=10)
<xarray.DataArray (y: 4)>
array([0.4515217, 0.54756005,
       0.506477, 0.73067568])
Coordinates:
    x        int64 10
  * y        (y) int64 10 20 30 40
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3
```

```
>>> data.y
<xarray.DataArray 'y' (y: 4)>
array([10, 20, 30, 40])
Coordinates:
  * y        (y) int64 10 20 30 40
Attributes:
    units:    cm

>>> data.sel(x=10,y=[30,40]).values
array([0.506477  , 0.73067568])

>>> data.sel(y=slice(15,30))     # only 15<=y<=30
<xarray.DataArray (y: 2, x: 3)>
array([[0.54756005, 0.21070062, 0.71370984],
       [0.506477  , 0.83279696, 0.1229428 ]])
Coordinates:
  * x        (x) int64 10 11 12
  * y        (y) int64 20 30
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3
```

DataArray.interp() to interpolate by coordinate value

```
>>> data
<xarray.DataArray (y: 4, x: 3)>
array([[0.4515217 , 0.73093109, 0.85287453],   [0.54756005, 0.21070062, 0.71370984],
       [0.506477  , 0.83279696, 0.1229428 ],   [0.73067568, 0.64027954, 0.999999  ]])
Coordinates:
  * x        (x) int64 10 11 12
  * y        (y) int64 10 20 30 40
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3

>>> data.interp(x=10.5, y=15)   # between 1st and 2nd rows, between 1st and 2nd columns
<xarray.DataArray ()>
array(0.48517836)
Coordinates:
    x        float64 10.5
    y        int64 15
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3
```

DataArray.interp() to interpolate by coordinate value

```
>>> data
<xarray.DataArray (y: 4, x: 3)>
array([[0.4515217 , 0.73093109, 0.85287453],   [0.54756005, 0.21070062, 0.71370984],
       [0.506477  , 0.83279696, 0.1229428 ],   [0.73067568, 0.64027954, 0.999999  ]])
Coordinates:
  * x        (x) int64 10 11 12
  * y        (y) int64 10 20 30 40
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3

>>> data.interp(x=10.5, y=15)   # between 1st and 2nd rows, between 1st and 2nd columns
<xarray.DataArray ()>
array(0.48517836)
Coordinates:
    x        float64 10.5
    y        int64 15
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3

>>> data.interp(x=10.5, y=15, method="nearest").values   # closest neighbour
array(0.4515217)
```

DataArray.interp() to interpolate by coordinate value

```
>>> data
<xarray.DataArray (y: 4, x: 3)>
array([[0.4515217 , 0.73093109, 0.85287453],   [0.54756005, 0.21070062, 0.71370984],
       [0.506477  , 0.83279696, 0.1229428 ],   [0.73067568, 0.64027954, 0.999999  ]])
Coordinates:
  * x        (x) int64 10 11 12
  * y        (y) int64 10 20 30 40
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3

>>> data.interp(x=10.5, y=15)   # between 1st and 2nd rows, between 1st and 2nd columns
<xarray.DataArray ()>
array(0.48517836)
Coordinates:
    x        float64 10.5
    y        int64 15
Attributes:
    author:   AR
    date:     2020-09-30
    name:     density
    units:    g/cm^3

>>> data.interp(x=10.5, y=15, method="nearest").values   # closest neighbour
array(0.4515217)

>>> data.interp(y=15).values         # between 1st and 2nd rows
array([0.49954087, 0.47081586, 0.78329218])
```

Aggregate functions

```
meanOfEachColumn = data.mean(dim='y')     # apply mean over y
spatialMean = data.mean(dim=['x','y'])    # apply mean over both x and y
spatialMean = data.mean()                 # same

>>> meanOfEachColumn
<xarray.DataArray (x: 3)>
array([0.55905861, 0.60367705, 0.67238154])
Coordinates:
  * x        (x) int64 10 11 12

>>> spatialMean
<xarray.DataArray ()>
array(0.61170573)
```

DataArray.groupby()

```
>>> columns = data.groupby("x")
>>> columns
DataArrayGroupBy, grouped over 'x'
3 groups with labels 10, 11, 12.
```

DataArray.groupby()

```
>>> columns = data.groupby("x")
>>> columns
DataArrayGroupBy, grouped over 'x'
3 groups with labels 10, 11, 12.
```

You can apply a function separately to each group:

```
>>> columns.map(lambda v: v.sum()/len(v))      # could use v.mean() too with the same result
<xarray.DataArray (x: 3)>
array([0.55905861, 0.60367705, 0.67238154])    # same result as in the previous slide!
Coordinates:
  * x         (x) int64 10 11 12
```

DataArray.groupby()

```
>>> columns = data.groupby("x")
>>> columns
DataArrayGroupBy, grouped over 'x'
3 groups with labels 10, 11, 12.
```

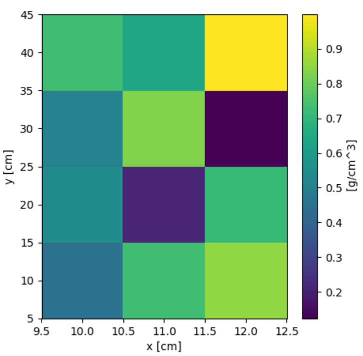
You can apply a function separately to each group:

```
>>> columns.map(lambda v: v.sum()/len(v))      # could use v.mean() too with the same result
<xarray.DataArray (x: 3)>
array([0.55905861, 0.60367705, 0.67238154])    # same result as in the previous slide!
Coordinates:
  * x          (x) int64 10 11 12

>>> columns.map(lambda v: (v-v.min())/(v.max()-v.min()))    # normalize data in each column
<xarray.DataArray (y: 4, x: 3)>
array([[0.        , 0.83625386, 0.83225194],
       [0.34403358, 0.        , 0.67357946],
       [0.19686375, 1.        , 0.        ],
       [1.        , 0.69053439, 1.        ]])
Coordinates:
  * x          (x) int64 10 11 12
  * y          (y) int64 10 20 30 40
```

Plotting

Matplotlib is integrated directly into xarray:

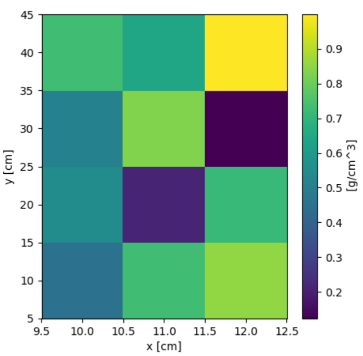
```
>>> data.plot(size=8)
<matplotlib.collections.QuadMesh object at 0x7fbf7a9c1a30>
>>> import matplotlib.pyplot as plt    # not needed inside Jupyter
>>> plt.show()                         # not needed inside Jupyter
```

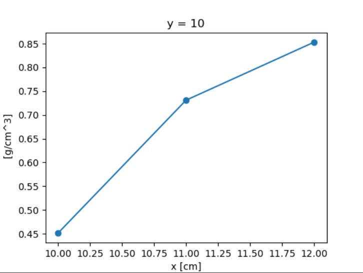


Plotting

Matplotlib is integrated directly into xarray:

```
>>> data.plot(size=8)
<matplotlib.collections.QuadMesh object at 0x7fbf7a9c1a30>
>>> import matplotlib.pyplot as plt   # not needed inside Jupyter
>>> plt.show()                        # not needed inside Jupyter
```






```
>>> data.isel(y=0).plot(marker="o", size=8, markersize=6)
[<matplotlib.lines.Line2D object at 0x7f80f4ca63d0>]
>>> plt.show()   # not needed inside Jupyter
```

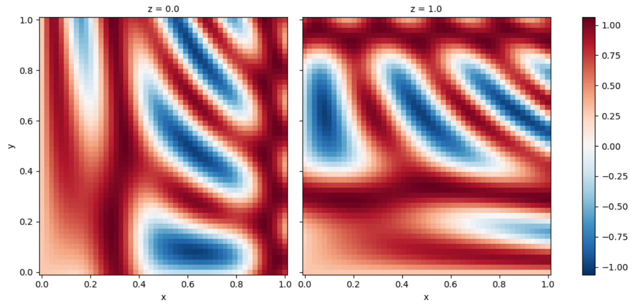
Create a 3D array

Let's create a function inside a unit cube $x, y, z \in [0, 1]$ on a 50^3 grid:

```
import numpy as np

n = 50
x = np.linspace(0, 1, n)
y = np.linspace(0, 1, n).reshape(n,1)
z = np.linspace(0, 1, n).reshape(n,1,1)

f1 = np.sin(5*x)**8 + np.cos(5+25*x*y)*np.cos(5*x)  # function at one side of the cube (z=0)
f2 = np.sin(5*y)**8 + np.cos(5+25*x*y)*np.cos(5*y)  # rotated 90 degrees, other sise (z=1)
f = (1-z)*f1 + z*f2

import xarray as xr
coords = {"z": z.flatten(), "y": y.flatten(), "x": x}    # supply 1D array to all
rho = xr.DataArray(
    f,
    dims=("z", "y", "x"),
    coords=coords
)
```

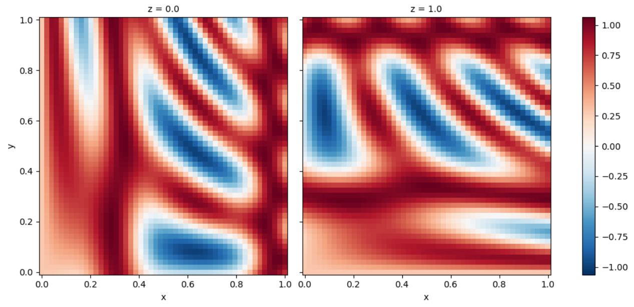
Plot this function

```
# rho.sel(z=[0,1]) is a 3D DataArray with two z values
# rho.sel(z=[0,1]).plot(size=8) will produce a histogram
rho.sel(z=[0,1]).plot(size=8, col="z")
import matplotlib.pyplot as plt    # not needed inside Jupyter
plt.show()                         # not needed inside Jupyter
```



Plot this function

```
# rho.sel(z=[0,1]) is a 3D DataArray with two z values
# rho.sel(z=[0,1]).plot(size=8) will produce a histogram
rho.sel(z=[0,1]).plot(size=8, col="z")
import matplotlib.pyplot as plt    # not needed inside Jupyter
plt.show()                         # not needed inside Jupyter
```




```
rho.to_netcdf("rho.nc")    # write to disk
```

⇒ let's view this function in ParaView!

Dataset

- *xarray.Dataset* is a collection of multiple *DataArray*'s that (usually) share dimensions
- Create a dataset from scratch, starting from the existing code for `rho` DataArray:

```
import numpy as np
n = 50
x = np.linspace(0, 1, n)
y = np.linspace(0, 1, n).reshape(n,1)
z = np.linspace(0, 1, n).reshape(n,1,1)
f1 = np.sin(5*x)**8 + np.cos(5+25*x*y)*np.cos(5*x) # function at one side of the cube (z=0)
f2 = np.sin(5*y)**8 + np.cos(5+25*x*y)*np.cos(5*y) # rotated 90 degrees, other sise (z=1)
f = (1-z)*f1 + z*f2
import xarray as xr
coords = {"z": z.flatten(), "y": y,flatten(), "x": x}    # supply 1D array to all
rho = xr.DataArray(f, dims=("z", "y", "x"), coords=coords)
```

```
temp = xr.DataArray(
     20 + 2*np.random.randn(n,n,n),    # standard normal distribution
     dims=("z", "y", "x"),
     coords=coords
)

ds = xr.Dataset({"temperature": temp, "density": rho,
                 "bar": ("x", 200+np.arange(n)), "pi": np.pi})
```

Dataset (cont.)

```
>>> ds
<xarray.Dataset>
Dimensions:       (x: 50, y: 50, z: 50)
Coordinates:
  * z             (z) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
  * y             (y) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
  * x             (x) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
Data variables:
    temperature  (z, y, x) float64 18.03 20.79 18.89 17.32 ... 16.97 22.71 18.51
    density      (z, y, x) float64 0.2837 0.2822 0.2778 ... 0.499 0.6163 0.7587
    bar          (x) int64 200 201 202 203 204 205 ... 244 245 246 247 248 249
    pi           float64 3.142
```

Dataset (cont.)

```
>>> ds
<xarray.Dataset>
Dimensions:        (x: 50, y: 50, z: 50)
Coordinates:
  * z              (z) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
  * y              (y) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
  * x              (x) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
Data variables:
    temperature    (z, y, x) float64 18.03 20.79 18.89 17.32 ... 16.97 22.71 18.51
    density        (z, y, x) float64 0.2837 0.2822 0.2778 ... 0.499 0.6163 0.7587
    bar            (x) int64 200 201 202 203 204 205 ... 244 245 246 247 248 249
    pi             float64 3.142

>>> ds.density.shape      # data.variables['density'].shape is equivalent
(50, 50, 50)
>>> ds.coords
Coordinates:
  * z          (z) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
  * y          (y) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
  * x          (x) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
>>> ds.pi
<xarray.DataArray 'pi' ()>
array(3.14159265)
```

Selecting floating values

```
>>> ds.x
<xarray.DataArray 'x' (x: 50)>
array([0.      , 0.020408, 0.040816, 0.061224, 0.081633, 0.102041, 0.122449,
       0.142857, 0.163265, 0.183673, 0.204082, 0.22449 , 0.244898, 0.265306,
       ...
       0.857143, 0.877551, 0.897959, 0.918367, 0.938776, 0.959184, 0.979592, 1.])
Coordinates:
  * x        (x) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0

>>> ds.x[1].values
array(0.02040816)
```

Selecting floating values

```
>>> ds.x
<xarray.DataArray 'x' (x: 50)>
array([0.      , 0.020408, 0.040816, 0.061224, 0.081633, 0.102041, 0.122449,
       0.142857, 0.163265, 0.183673, 0.204082, 0.22449 , 0.244898, 0.265306,
       ...
       0.857143, 0.877551, 0.897959, 0.918367, 0.938776, 0.959184, 0.979592, 1.])
Coordinates:
  * x        (x) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0

>>> ds.x[1].values
array(0.02040816)

>>> ds.density.sel(x=0.02040816)     # this likely won't work!
...
KeyError: 0.02040816
```

Selecting floating values

```
>>> ds.x
<xarray.DataArray 'x' (x: 50)>
array([0.      , 0.020408, 0.040816, 0.061224, 0.081633, 0.102041, 0.122449,
       0.142857, 0.163265, 0.183673, 0.204082, 0.22449 , 0.244898, 0.265306,
       ...
       0.857143, 0.877551, 0.897959, 0.918367, 0.938776, 0.959184, 0.979592, 1.])
Coordinates:
  * x        (x) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0

>>> ds.x[1].values
array(0.02040816)

>>> ds.density.sel(x=0.02040816)     # this likely won't work!
...
KeyError: 0.02040816

>>> ds.density.sel(x=ds.x[1])
<xarray.DataArray 'density' (z: 50, y: 50)>
array([[0.28218669, 0.29210391, 0.30198946, ..., 0.69748965, 0.70483736, 0.71210865],
       [0.2822168 , 0.29210391, 0.30189318, ..., 0.70429482, 0.7108887 , 0.71631057],
       ...
       [0.28366219, 0.29210391, 0.29727155, ..., 1.03094303, 1.00135312, 0.91800259]])
Coordinates:
  * z        (z) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
  * y        (y) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
    x        float64 0.02041
```

Selecting floating values

```
>>> ds.x
<xarray.DataArray 'x' (x: 50)>
array([0.      , 0.020408, 0.040816, 0.061224, 0.081633, 0.102041, 0.122449,
       0.142857, 0.163265, 0.183673, 0.204082, 0.22449 , 0.244898, 0.265306,
       ...
       0.857143, 0.877551, 0.897959, 0.918367, 0.938776, 0.959184, 0.979592, 1.])
Coordinates:
  * x        (x) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0

>>> ds.x[1].values
array(0.02040816)

>>> ds.density.sel(x=0.02040816)     # this likely won't work!
...
KeyError: 0.02040816

>>> ds.density.sel(x=ds.x[1])
<xarray.DataArray 'density' (z: 50, y: 50)>
array([[0.28218669, 0.29210391, 0.30198946, ..., 0.69748965, 0.70483736, 0.71210865],
       [0.2822168 , 0.29210391, 0.30189318, ..., 0.70429482, 0.7108887 , 0.71631057],
       ...
       [0.28366219, 0.29210391, 0.29727155, ..., 1.03094303, 1.00135312, 0.91800259]])
Coordinates:
  * z        (z) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
  * y        (y) float64 0.0 0.02041 0.04082 0.06122 ... 0.9592 0.9796 1.0
    x        float64 0.02041
```

Or you can use `ds.density.isel(x=1)`
or `ds.density.sel(x=slice(0.01,0.03))`

Save the dataset to disk

```
ds.to_netcdf("cube.nc")
```

$\Rightarrow$ let's view this function in ParaView!

Before hitting apply, select Dimensions: (z,y,x)

Write to NetCDF in single precision (save disk space!)

Start with single-precision numpy arrays:

```
import numpy as np
import xarray as xr
n = 50
x = np.linspace(0, 1, n, dtype=np.float32)
y = np.linspace(0, 1, n, dtype=np.float32).reshape(n,1)
z = np.linspace(0, 1, n, dtype=np.float32).reshape(n,1,1)
f1 = np.sin(5*x)**8 + np.cos(5+25*x*y)*np.cos(5*x)
f2 = np.sin(5*y)**8 + np.cos(5+25*x*y)*np.cos(5*y)
f = (1-z)*f1 + z*f2
coords = {"z": z.flatten(), "y": y.flatten(), "x": x}
rho = xr.DataArray(f, dims=("z", "y", "x"), coords=coords)
temp = xr.DataArray(20 + 2*np.float32(np.random.randn(n,n,n)),
                    dims=("z", "y", "x"), coords=coords)
ds = xr.Dataset({"temperature": temp, "density": rho,
                 "bar": ("x", 200+np.arange(n,dtype=np.float32)), "pi": np.float32(np.pi)})

print(ds.density.dtype, ds.temperature.dtype, ds.bar.dtype, ds.pi.dtype)
          # float32 float32 float32 float32

ds.to_netcdf("cubeSinglePrecision.nc")
```

Dataset in spherical geometry

To store the file in a NetCDF Climate and Forecast (CF) convention, replace `x,y,z` with `lat,lon,r` (exact names not important) and use `ds.lat.attrs["units"] = "degrees_north"` and `ds.lon.attrs["units"] = "degrees_east"`:

```
from numpy import linspace, sin, cos, float32, radians, random
import xarray as xr
nlat, nlon, nr = 181, 361, 30    # grid resolution
lat = linspace(-90, 90, nlat, dtype=float32)
lon = linspace(0,  360, nlon, dtype=float32).reshape(nlon,1)
r =   linspace(0.9,  1,    nr, dtype=float32).reshape(nr,1,1)
f = (1+0.8*cos(radians(lon))*cos(radians(lat))) * sin(radians(lat))**4 * r
coords = {"lat": lat, "lon": lon.flatten(), "r": r.flatten()}
rho = xr.DataArray(f, dims=("r", "lon", "lat"), coords=coords)
temp = xr.DataArray(20 + 2*float32(random.randn(nr,nlon,nlat)),
                    dims=("r", "lon", "lat"), coords=coords)
ds = xr.Dataset({"temperature": temp, "density": rho})
ds.lat.attrs["units"] = "degrees_north"    # important!
ds.lon.attrs["units"] = "degrees_east"     # important!
ds.to_netcdf("spherical.nc")
```

$\Rightarrow$ let's load it into ParaView!

Time series data

Xarray relies on pandas functions for time-series functionality:

```
>>> import pandas as pd
>>> time = pd.date_range("2000-01-01", freq="D", periods=365*3+1)     # 2000-Jan to 2002-Dec
>>> time
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-05', '2000-01-06', '2000-01-07', '2000-01-08',
               ...
               '2002-12-30', '2002-12-31'],
              dtype='datetime64[ns]', length=1096, freq='D')
```

Time series data

Xarray relies on pandas functions for time-series functionality:

```
>>> import pandas as pd
>>> time = pd.date_range("2000-01-01", freq="D", periods=365*3+1)     # 2000-Jan to 2002-Dec
>>> time
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-05', '2000-01-06', '2000-01-07', '2000-01-08',
               ...
               '2002-12-30', '2002-12-31'],
              dtype='datetime64[ns]', length=1096, freq='D')

>>> time.month
Int64Index([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
            ...
            12, 12, 12, 12, 12, 12, 12, 12, 12, 12],
           dtype='int64', length=1096)
```

Time series data

Xarray relies on pandas functions for time-series functionality:

```
>>> import pandas as pd
>>> time = pd.date_range("2000-01-01", freq="D", periods=365*3+1)     # 2000-Jan to 2002-Dec
>>> time
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-05', '2000-01-06', '2000-01-07', '2000-01-08',
               ...
               '2002-12-30', '2002-12-31'],
              dtype='datetime64[ns]', length=1096, freq='D')

>>> time.month
Int64Index([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
            ...
            12, 12, 12, 12, 12, 12, 12, 12, 12, 12],
           dtype='int64', length=1096)

>>> time.day
Int64Index([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10,
            ...
            22, 23, 24, 25, 26, 27, 28, 29, 30, 31],
           dtype='int64', length=1096)
```

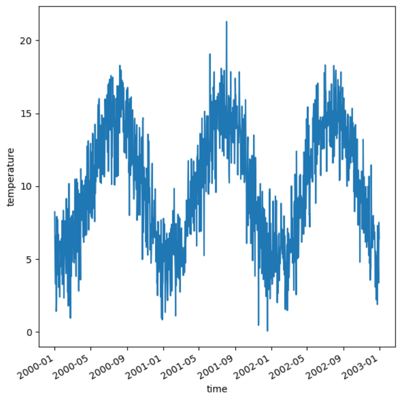
You can always type `help(pd.date_range)` or `help(time)`

Time-dependent scalar dataset

```
import xarray as xr
import numpy as np
ntime = len(time)
temp = 10 + 5*np.sin((250+np.arange(ntime))/365.25*2*np.pi) + 2*np.random.randn(ntime)
ds = xr.Dataset({ "temperature": ("time", temp), "time": time })      # 1D function of time
ds.temperature.plot(size=5)
import matplotlib.pyplot as plt    # not needed inside Jupyter
plt.show()                         # not needed inside Jupyter
```

Time-dependent scalar dataset

```
import xarray as xr
import numpy as np
ntime = len(time)
temp = 10 + 5*np.sin((250+np.arange(ntime))/365.25*2*np.pi) + 2*np.random.randn(ntime)
ds = xr.Dataset({ "temperature": ("time", temp), "time": time })      # 1D function of time
ds.temperature.plot(size=5)
import matplotlib.pyplot as plt    # not needed inside Jupyter
plt.show()                         # not needed inside Jupyter
```




```
>>> ds
<xarray.Dataset>
Dimensions:       (time: 1096)
Coordinates:
  * time          (time) datetime64[ns] 2000-01-01 2000-01-02 ... 2002-12-31
Data variables:
    temperature   (time) float64 8.235 5.789 4.279 3.287 ... 7.515 6.399
```

Time subsetting

```
>>> ds.isel(time=-1)   # last timestep
<xarray.Dataset>
Dimensions:      ()
Coordinates:
    time         datetime64[ns] 2002-12-31
Data variables:
    temperature  float64 6.399

>>> ds.isel(time=-1).temperature
<xarray.DataArray 'temperature' ()>
array(6.39868414)
Coordinates:
    time     datetime64[ns] 2002-12-31
```

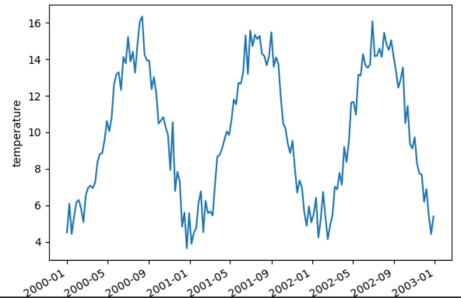
Time subsetting

```
>>> ds.isel(time=-1)   # last timestep
<xarray.Dataset>
Dimensions:      ()
Coordinates:
    time         datetime64[ns] 2002-12-31
Data variables:
    temperature  float64 6.399

>>> ds.isel(time=-1).temperature
<xarray.DataArray 'temperature' ()>
array(6.39868414)
Coordinates:
    time     datetime64[ns] 2002-12-31

>>> ds.sel(time="2002-12-22").temperature
<xarray.DataArray 'temperature' ()>
array(5.24044602)
Coordinates:
    time     datetime64[ns] 2002-12-22
```

Time subsetting

```
>>> ds.isel(time=-1)    # last timestep
<xarray.Dataset>
Dimensions:      ()
Coordinates:
    time         datetime64[ns] 2002-12-31
Data variables:
    temperature  float64 6.399

>>> ds.isel(time=-1).temperature
<xarray.DataArray 'temperature' ()>
array(6.39868414)
Coordinates:
    time     datetime64[ns] 2002-12-31


>>> ds.sel(time="2002-12-22").temperature
<xarray.DataArray 'temperature' ()>
array(5.24044602)
Coordinates:
    time     datetime64[ns] 2002-12-22
```

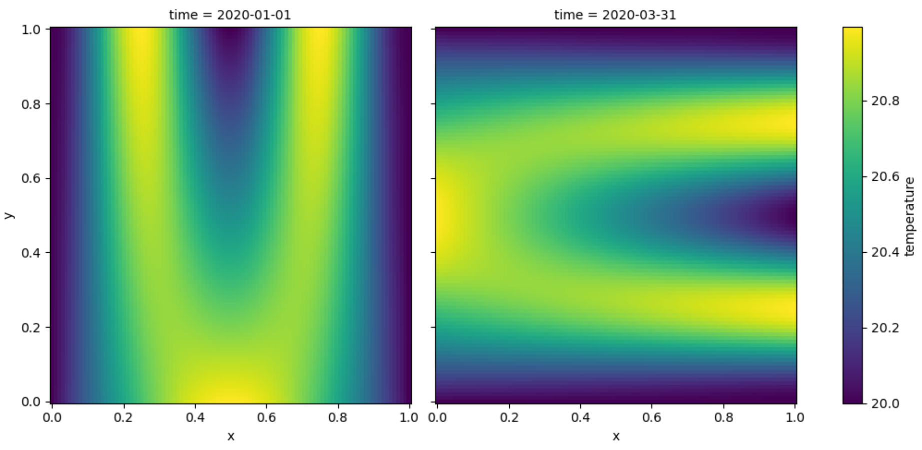
```
>>> ds.resample(time='7D')
# from 1096 steps to 157 time groups
DatasetResample, grouped over '__resample_dim__'
157 groups with labels 2000-01-01, ..., 2002-12-28.

>>> weekly = ds.resample(time='7D').mean()
# compute mean for each group
>>> weekly.temperature.plot(size=8)
[<matplotlib.lines.Line2D object at 0x7fc8b8db3f70>]
>>> plt.show()      # not needed inside Jupyter
```



Take our 3D dataset (recall `cube.nc`) and make it 2D+time:

```
import numpy as np
import pandas as pd
import xarray as xr

time = pd.date_range("2020-01-01", freq="D", periods=91) # January - March 2020
ntime = len(time)
n = 100       # spatial resolution in each dimension
axis = np.linspace(0,1,n)
X, Y = np.meshgrid(axis,axis)    # 2D Cartesian meshes of x,y coordinates
initialState = (1-Y)*np.sin(np.pi*X) + Y*(np.sin(2*np.pi*X))**2
finalState =   (1-X)*np.sin(np.pi*Y) + X*(np.sin(2*np.pi*Y))**2
f = np.zeros((ntime,n,n))
for t in range(ntime):
    z = (t+0.5) / ntime    # dimensionless time from 0 to 1
    f[t,:,:] = (1-z)*initialState + z*finalState

coords = {"time": time, "x": axis, "y": axis}
temp = xr.DataArray(    # 2D array varying in time from initialState to finalState
    20 + f, dims=("time","y","x"), coords=coords
)
pres = xr.DataArray(    # random 2D array
    100 + 10*np.random.randn(ntime,n,n), dims=("time","y","x"), coords=coords
)
ds = xr.Dataset({"temperature": temp, "pressure": pres})
ds.to_netcdf("evolution.nc")
```

```
ds.isel(time=[0,-1]).temperature.plot(size=5,col="time")    # first and last steps
import matplotlib.pyplot as plt    # not needed inside Jupyte
plt.show()                         # not needed inside Jupyter
```



⇒ let's play back `evolution.nc` in ParaView!

Earth's mantle convection

- Upcoming SciVis competition, will be announced towards the end of October
  - open to anyone (no research affiliation necessary)
  - please keep an eye on our emails and `https://www.westgrid.ca`
- Data courtesy of Hosein Shahnas and Russell Pysklywec at the UofToronto
  - simulation conducted using Compute Canada's Niagara cluster (SciNet)
  - each timestep saved as a separate multi-variable NetCDF file on a spherical grid

Live demo

Questions?